Estruturas de Dados

Aplicações e Recursividade

Aula 05

Prof. Felipe A. Louza

Roteiro

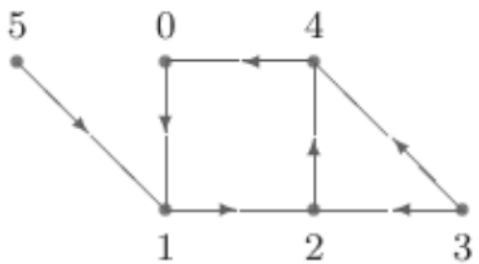
Roteiro

Aplicação de Filas - Distância entre cidades

Considere N cidades numeradas de 0 a N-1 interligadas por estradas de mão única[1].

- As ligações entre as cidades são representadas por uma matriz `A[N][N]` da seguinte maneira:
  - $A[i][j]$ `vale 1` se existe estrada de i para j, ou 0 caso contrário.

	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	1	0	0	0
2	0	0	0	0	1	0
3	0	0	1	0	1	0
4	1	0	0	0	0	0
5	0	1	0	0	0	0



Como calcular a (menor) distância de $i \rightsquigarrow j$??

[1] Vamos supor que as distâncias tem tamanho 1

Cliente

distancias.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "lib/fila.h"
```

```c
int main(){
  int n, e, i, j;
  scanf("%d %d", &n, &e);
  int **M = alocar_matriz(n, n);
  ler_matriz(M, e);
  int **dist = alocar_matriz(n, n);
  for(i=0; i<n; i++) //calcular distancias
    distancia(M, n, dist[i], i);
  imprimir_matriz(dist, n, n);
  desalocar_matriz(dist, n);
  desalocar_matriz(M, n);
return 0;
}
```

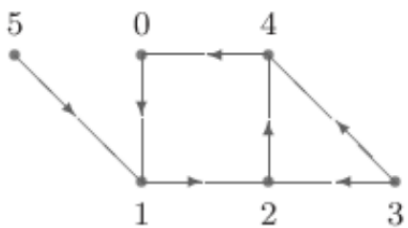
Cliente

distancias.c

```
32 int** alocar_matriz(int n, int e){
33   int i, j;
34   int **M = (int**) malloc(n*sizeof(int*));
35   for(i=0; i<n; i++){
36     M[i] = (int*) malloc(n*sizeof(int));
37     for(j=0; j<n; j++)
38       M[i][j] = 0;
39   }
40 return M;
41 }
```

	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	1	0	0	0
2	0	0	0	0	1	0
3	0	0	1	0	1	0
4	1	0	0	0	0	0
5	0	1	0	0	0	0

```
43 void ler_matriz(int **M, int e){ //recebe M
44   int i, j, x, y;
45   for(i=0; i<e; i++){
46     scanf("%d %d", &x, &y);
47     M[x][y] = 1;
48   }
49 }
```



Cliente

distancias.c

```
51 void  desalocar_matriz(int** M, int l){ //recebe M
52   int i;
53   for(i=0; i<l; i++) free(M[i]);
54   free(M);
55 }
56
57 void  imprimir_matriz(int** M, int l, int c){
58   int i,j;
59   for(i=0; i<l; i++){
60     for(j=0; j<c; j++) printf("%d ", M[i][j]);
61     printf("\n");
62   }
63 }
```

	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	1	0	0	0
2	0	0	0	0	1	0
3	0	0	1	0	1	0
4	1	0	0	0	0	0
5	0	1	0	0	0	0

Aplicação de Filas - Distância entre cidades

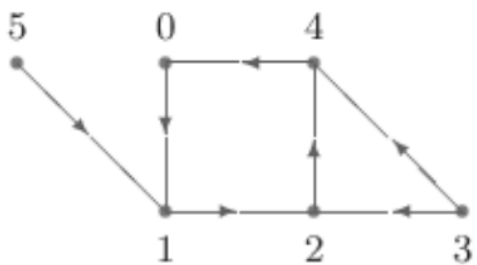
Nosso problema:

- Dada uma cidade `c`, determinar a distância de `c` para cada uma das demais cidades.
- A distância de uma cidade `c` a uma cidade `i` é o menor número de estradas para ir de `c` a `i`.

As distâncias podem ser armazenadas em um vetor `dist`

- `dist[i]` é a distância de $c \rightsquigarrow i$.
- Se for impossível ir de `c` a `i`, então `dist[i]=N`.

i	0	1	2	3	4	5
dist[i]	2	3	1	0	1	6

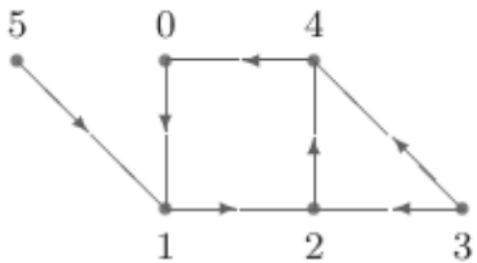


Aplicação de Filas - Distância entre cidades

Algoritmo: fila de cidades ativas.

- Passo 1: insira `c` na Fila.
- Passo 2: Remove o topo da Fila em `i`, e insere as cidade vizinhas à `i` que ainda não foram visitadas na Fila (ex. j_1, j_2 e j_3). Além disso, $dist[j_i] = dist[i] + 1$
- Passo 3: Repita o passo 2 enquanto a Fila não estiver vazia.

i	0	1	2	3	4	5
dist[i]	6	6	6	0	6	6

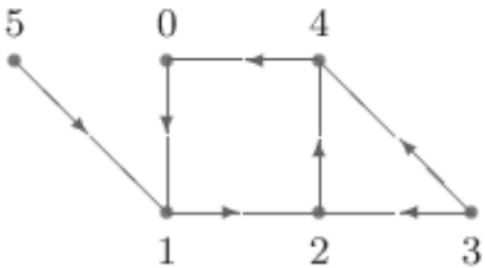


Cidades ainda não visitadas: `dist[i] = N`.

Cliente

distancias.c

```
67 void distancia(int **M, int n, int *dist, int c){
68   int i, j;
69   for(i = 0; i < n; i++) dist[i] = n;
70   dist[c] = 0;
71
72   Fila *F = fila_criar();
73   fila_adicionar(&F, c);
74
75   while(fila_tamanho(F)>0){
76     i = fila_topo(F);
77     fila_remover(&F);
78     for(j = 0; j < n; j++){
79       if(M[i][j] == 1 && dist[j] >= n) {//i->j?
80         dist[j] = dist[i] + 1;
81         fila_adicionar(&F, j);
82       }
83     }
84   }
85   fila_destruir(&F);
86 }
```



i	0	1	2	3	4	5
dist[i]	6	6	6	0	6	6

Makefile

Vamos usar o Makefile para compilar:

```
1  CFLAGS= -Wall -Werror
2  LIB= ./lib
3
4  all: distancias
5
6  distancias: distancias.c $(LIB)/fila.o $(LIB)/lista_circular.o
7          gcc $^ -o $@
8
9  #regra genérica
10 %.o: %.c %.h
11         gcc $(CFLAGS) -c $< -o $@
```

Algumas novidades:

- Adicionamos os TADs no diretório `LIB=./lib`

`$^` representa todas as dependências da regra, `$@` o alvo, e `$<` representa a primeira dependência.

Roteiro

Aplicação de Pilhas - Parênteses balanceados

Considere o problema de decidir se uma dada sequência de parênteses está bem-formada (balanceada).

- Exemplos corretos:
  1. $(a + b)$
  2. $(a/(b + c))$
  3. $(a * b) + (c/(d - e))$
- Exemplos incorretos:
  4. $(a + b$
  5. $(a \cdot b) + (c/d - e))$
  6. $)a + b($

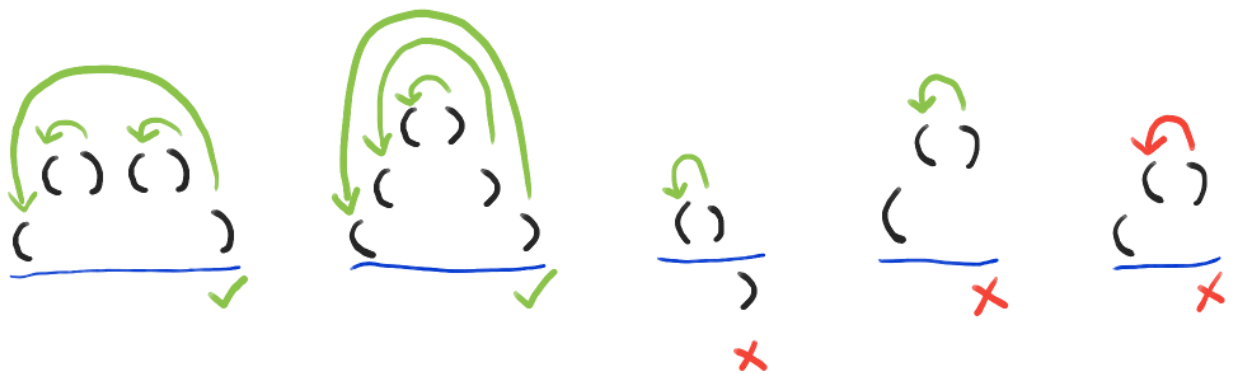
Nosso problema:

- Escreva uma função que, dada uma **sequência de parênteses**, diz se ela é balanceada ou não
  - Vamos ignorar operandos e operadores
  - ()(())

Aplicação de Pilhas - Parênteses balanceados

Uma sequência de parênteses S é balanceada se for

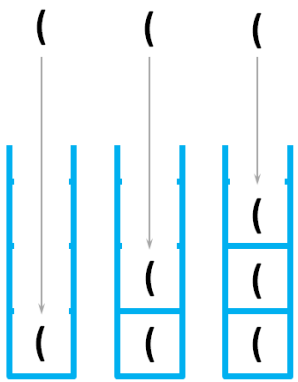
- vazia ()
- ou (sequência balanceada)
- ou a concatenação de duas sequências balanceadas (S)(S)



Aplicação de Pilhas - Parênteses balanceados

Algoritmo: para testar, leia da esquerda para a direita cada símbolo e se:

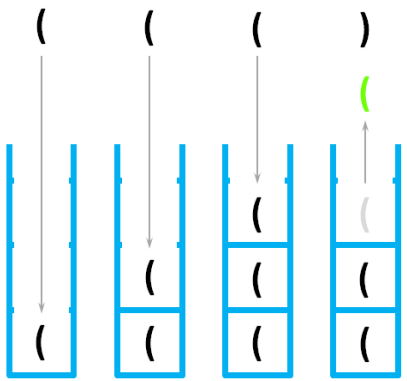
1. leu (: empilha o símbolo lido
2. leu): verifique se o topo da pilha é (, se sim, desempilhe o topo, caso contrário, retorne `FALSE`



Aplicação de Pilhas - Parênteses balanceados

Algoritmo: para testar, leia da esquerda para a direita cada símbolo e se:

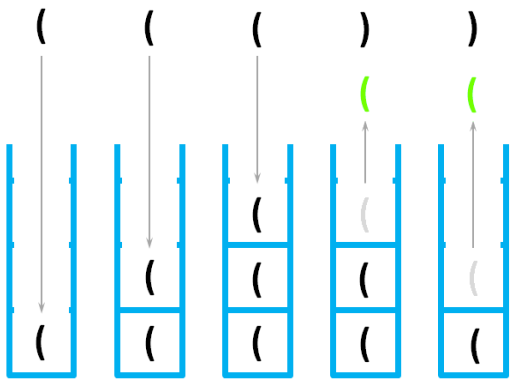
1. leu (: empilha o símbolo lido
2. leu): verifique se o topo da pilha é (, se sim, desempilhe o topo, caso contrário, retorne `FALSE`



Aplicação de Pilhas - Parênteses balanceados

Algoritmo: para testar, leia da esquerda para a direita cada símbolo e se:

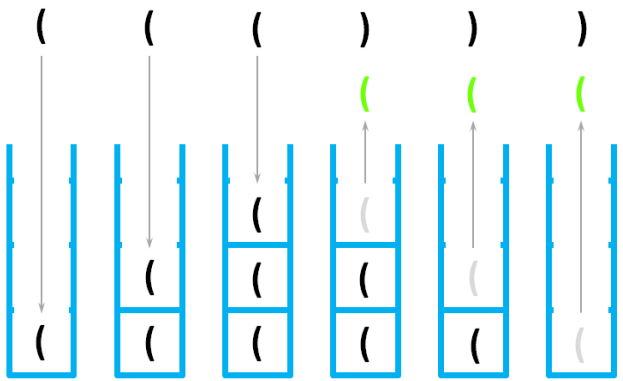
1. leu (: empilha o símbolo lido
2. leu): verifique se o topo da pilha é (, se sim, desempilhe o topo, caso contrário, retorne `FALSE`



Aplicação de Pilhas - Parênteses balanceados

Algoritmo: para testar, leia da esquerda para a direita cada símbolo e se:

1. leu (: empilha o símbolo lido
2. leu): verifique se o topo da pilha é (, se sim, desempilhe o topo, caso contrário, retorne `FALSE`

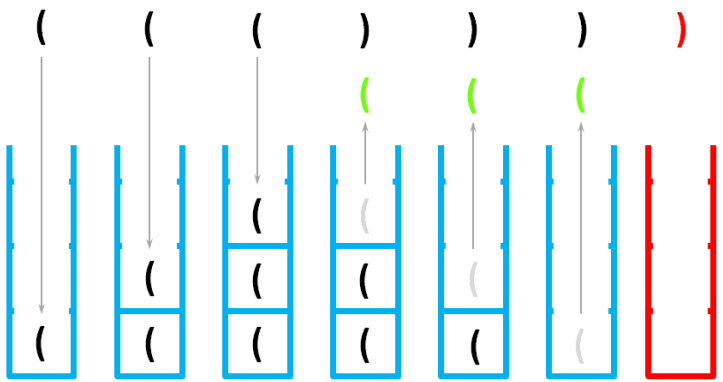


Se no final a Pilha estiver vazia, retorne `TRUE`.

Aplicação de Pilhas - Parênteses balanceados

Algoritmo: para testar, leia da esquerda para a direita cada símbolo e se:

1. leu (: empilha o símbolo lido
2. leu): verifique se o topo da pilha é (, se sim, desempilhe o topo, caso contrário, retorne `FALSE`

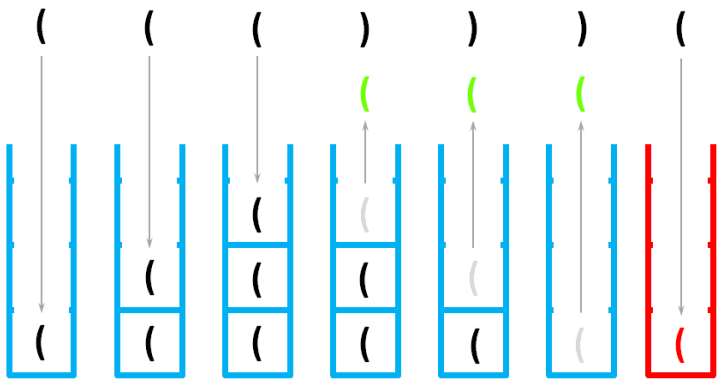


Se no final a Pilha estiver vazia, retorne `TRUE`.

Aplicação de Pilhas - Parênteses balanceados

Algoritmo: para testar, leia da esquerda para a direita cada símbolo e se:

1. leu (: empilha o símbolo lido
2. leu): verifique se o topo da pilha é (, se sim, desempilhe o topo, caso contrário, retorne `FALSE`



Se no final a Pilha estiver vazia, retorne `TRUE`.

Cliente

parenteses.c

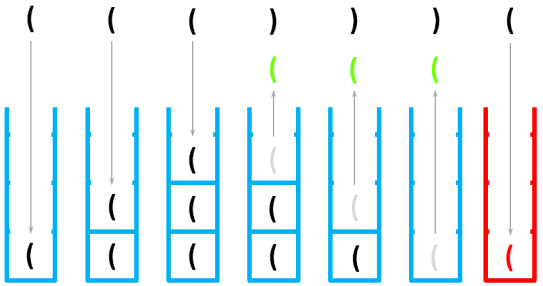
```c
#include <stdio.h>
#include <string.h>
#include "lib/pilha.h"
```

```c
int main(int argc, char **argv){
  if(argc!=2){
    printf("usage: %s \"sequence\"\n", argv[0]);
    return 1;
  }
  printf("%s\n", argv[1]);
  if(balanceada(argv[1])) printf("SIM\n");
  else printf("NAO\n");
return 0;
}
```

Cliente

parenteses.c

```
24 int balanceada(char *s){
25   int i, ok = 1;
26   int n = strlen(s);
27   Pilha *P = pilha_criar();
28   for (i = 0; i < n && ok; i++){
29     if (s[i] == '(')
30       pilha_adicionar(&P, s[i]);
31     else if (s[i] == ')') {
32       if (pilha_tamanho(P)==0) ok = 0;
33       else{
34         char c = pilha_topo(P);
35         pilha_remover(&P);
36       }
37     }
38   }
39   if (pilha_tamanho(P)>0) ok = 0;
40   pilha_destruir(&P);
41 return ok;
42 }
```



Makefile

Vamos usar o Makefile para compilar:

```
1 parenteses: parenteses.c $(LIB)/pilha.o $(LIB)/lista_com_cabeca.o
2   gcc $^ -o $@
```

Vamos executar:

```
1 ./parenteses "()()()"
2 ()()()
3 SIM
```

Makefile

Vamos usar o Makefile para compilar:

```
1 parenteses: parenteses.c $(LIB)/pilha.o $(LIB)/lista_com_cabeca.o
2   gcc $^ -o $@
```

Vamos executar:

```
1 ./parenteses "()()()"
2 ()()()
3 SIM
```

```
1 ./parenteses "(())"
2 (())
3 SIM
```

Makefile

Vamos usar o Makefile para compilar:

```
1 parenteses: parenteses.c $(LIB)/pilha.o $(LIB)/lista_com_cabeca.o
2   gcc $^ -o $@
```

Vamos executar:

```
1 ./parenteses "()()()"
2 ()()()
3 SIM
```

```
1 ./parenteses "(())"
2 (())
3 SIM
```

```
1 ./parenteses "()()())"
2 ()()())
3 NAO
```

Roteiro

Aplicação de Pilhas - Notação posfixa

Em expressões aritméticas, usalmente os operadores são escritos entre os operandos na forma `a + b`, conhecida como notação infixa:

- Precisamos definir a ordem de precedência entre os operadores para calcular `a + b * c`
- Essa ordem pode ser modificada com parênteses `(a + b) * c`

Aplicação de Pilhas - Notação posfixa

Em expressões aritméticas, usalmente os operadores são escritos entre os operandos na forma `a + b`, conhecida como notação infixa:

- Precisamos definir a ordem de precedência entre os operadores para calcular `a + b * c`
- Essa ordem pode ser modificada com parênteses `(a + b) * c`

Já na notação pósfixa[2], o operador é colocado após os seus dois operandos.

```
a b +
```

- A ordem dos operadores diz a ordem em que eles vão ser executados (da esquerda para a direita) `a b c + *`
- Não é necessário utilizar parênteses

[2] Também conhecida como notação polonesa inversa

Aplicação de Pilhas - Notação posfixa

Alguns exemplos:

infixa	posfixa
A+B*C	ABC*+
A*(B+C)/D-E	ABC+*D/E-
A+B*C/D*E-F	ABC*D/E*+F-
A+B+C*D-E*F*G	AB+CD*+EF*G*-
A+(B-(C+(D-(E+F))))	ABCDEF+-+-+

Aplicação de Pilhas - Notação posfixa

Alguns exemplos:

infixa	posfixa
`A+B*C`	`ABC*+`
`A*(B+C)/D-E`	`ABC+*D/E-`
`A+B*C/D*E-F`	`ABC*D/E*+F-`
`A+B+C*D-E*F*G`	`AB+CD*+EF*G*-`
`A+(B-(C+(D-(E+F))))`	`ABCDEF+-+-+`

Observe que os operandos (`A`, `B`, `C`, etc.) aparecem na mesma ordem na expressão infixa e na correspondente expressão posfixa.

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a}\ \texttt{b}\ \oplus \longrightarrow \texttt{a}\ \oplus\ \texttt{b}$$

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

2

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

`2 2`

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

`2 2 1`

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

`2 2 1 +`

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

`2 3`

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

```
2 3 4
```

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

`2 3 4 *`

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

`2 12`

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

`2 12 1`

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

`2 12 1 +`

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

`2 13`

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

`2 13 *`

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

26

Aplicação de Pilhas - Notação posfixa

Nosso problema:

- Calcule o resultado de uma expressão na forma posfixa

Ideia da solução:

- Sempre que um operador $\oplus$ é visto, aplicamos $\oplus$ nos dois últimos operandos vistos `a` e `b`

$$\texttt{a b} \oplus \longrightarrow \texttt{a} \oplus \texttt{b}$$

Exemplo:

- Infixa: `2 * ((2 + 1) * 4 + 1)` $= 26$
- Pósfixa: `2 2 1 + 4 * 1 + *`

26

Vamos utilizar uma pilha para recuperar os operandos mais recentes

Aplicação de Pilhas - Notação posfixa

Algoritmo: pilha de operandos

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

```
2 2 1 + 4 * 1 + *
```

Cliente

posfixa.c

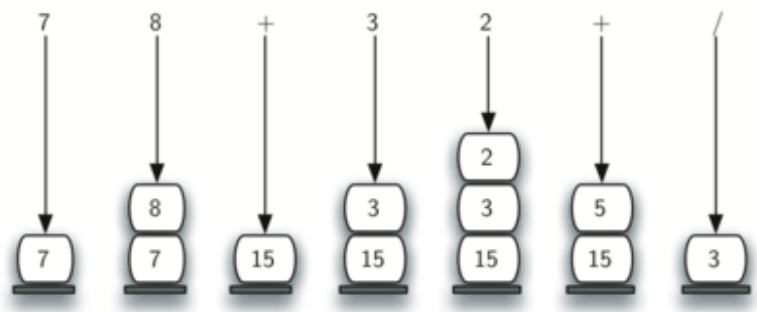
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lib/pilha.h"
```

```c
int main(int argc, char **argv){
  if(argc<2){
    printf("usage: %s \"expression\"\n", argv[0]);
    return 1;
  }
  printf("= %d\n", posfixa(&argv[1], argc-1));
return 0;
}
```

Cliente

posfixa.c

```
24 int posfixa(char **s, int n){
25   int result=0, i;
26   Pilha *P = pilha_criar();
27   for (i = 0; i < n; i++){
28     printf("%s ", s[i]);
29     if(s[i][0]=='+' || s[i][0]=='-' || s[i][0]=='/' || s[i][0]=='*'){
30       int b = pilha_topo(P);
31       pilha_remover(&P);
32       int a = pilha_topo(P);
33       pilha_remover(&P);
34       pilha_adicionar(&P, operacao(a, b, s[i]));
35     }
36     else{
37       pilha_adicionar(&P, atoi(s[i]));
38     }
39   }
40   result = pilha_topo(P);
41   pilha_destruir(&P);
42 return result;
43 }
```



Cliente

posfixa.c

```
45 int operacao(int a, int b, char *op){
46   switch(op[0]){
47     case '+':
48       return a+b;
49     case '-':
50       return a-b;
51     case '*':
52       return a*b;
53     case '/':
54       return a/b;
55     default:
56       return 0;
57   }
58 }
```

Vamos assumir que a expressão é bem formada

Makefile

Vamos usar o Makefile para compilar:

```
1 posfixa: posfixa.c $(LIB)/pilha.o $(LIB)/lista_com_cabeca.o
2   gcc $^ -o $@
```

Vamos executar:

```
1 ./posfixa 1 3 +
2 1 3 + = 4
```

Makefile

Vamos usar o Makefile para compilar:

```
1 posfixa: posfixa.c $(LIB)/pilha.o $(LIB)/lista_com_cabeca.o
2   gcc $^ -o $@
```

Vamos executar:

```
1 ./posfixa 1 3 +
2 1 3 + = 4
```

```
1 ./posfixa 7 8 + 3 2 + /
2 7 8 + 3 2 + / = 3
```

Makefile

Vamos usar o Makefile para compilar:

```
1 posfixa: posfixa.c $(LIB)/pilha.o $(LIB)/lista_com_cabeca.o
2   gcc $^ -o $@
```

Vamos executar:

```
1 ./posfixa 1 3 +
2 1 3 + = 4
```

```
1 ./posfixa 7 8 + 3 2 + /
2 7 8 + 3 2 + / = 3
```

```
1 ./posfixa 2 2 1 + 4 "*" 1 + "*"
2 2 2 1 + 4 * 1 + * = 26
```

Roteiro

Pilhas e recursão

Pergunta: Qual a relação entre pilhas e recursão?

Pilhas e recursão

Pergunta: Qual a relação entre pilhas e recursão?

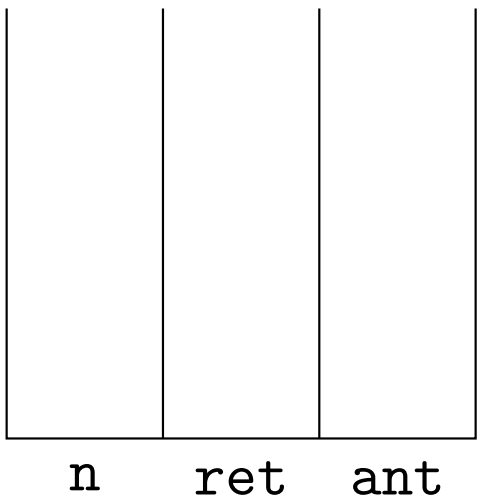
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```

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

Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

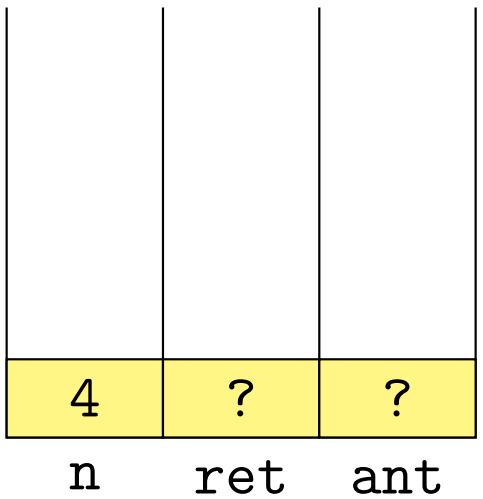
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

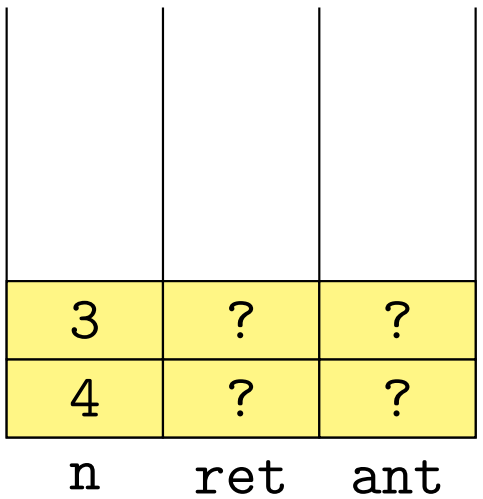
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

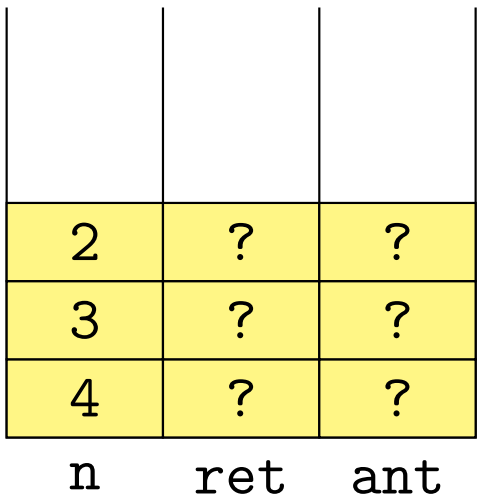
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

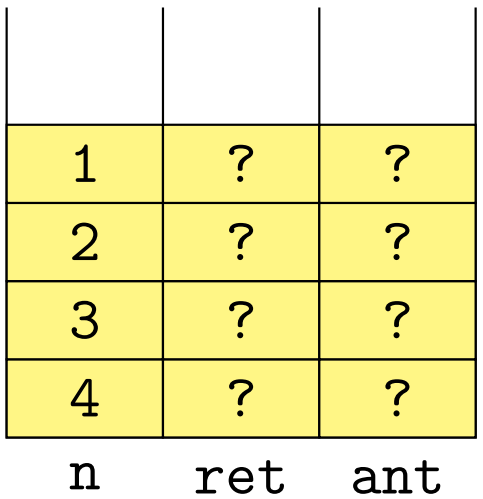
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

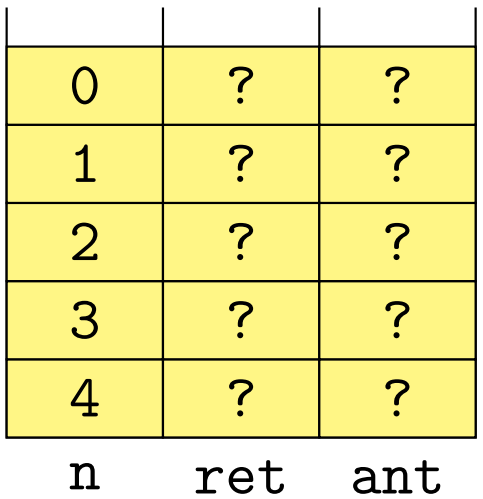
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

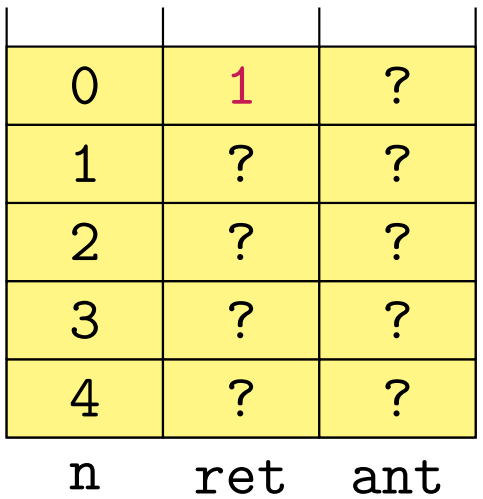
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

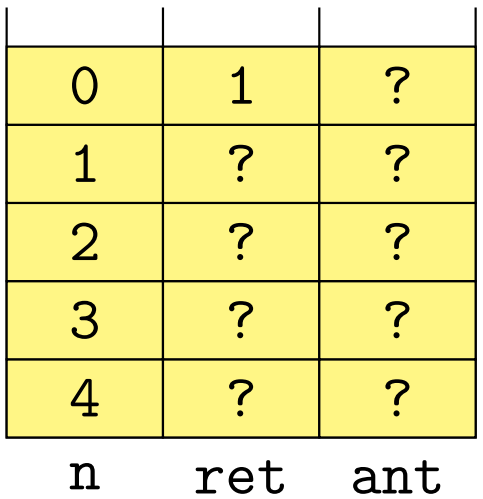
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

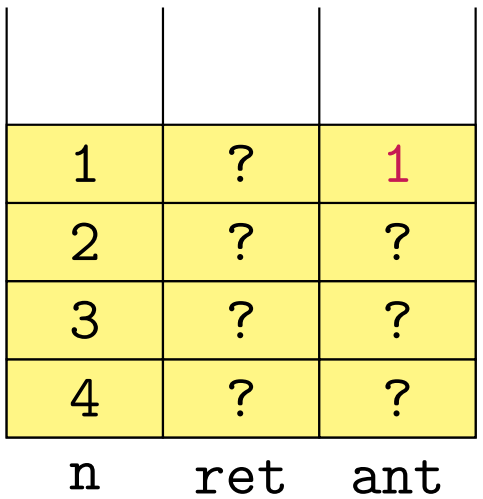
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

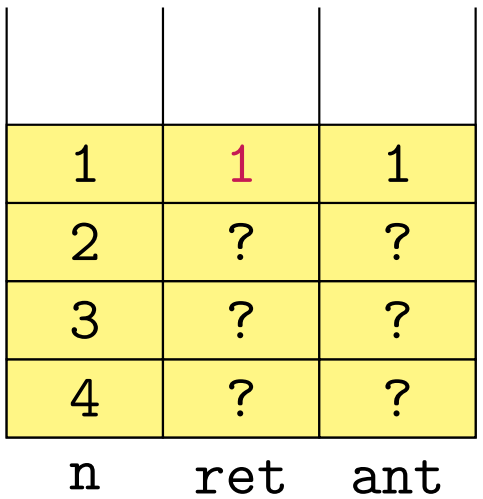
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

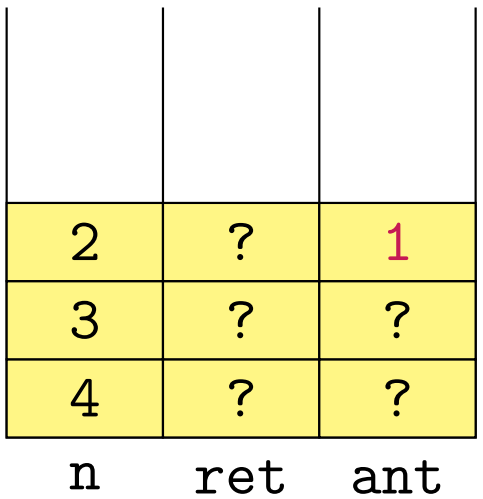
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

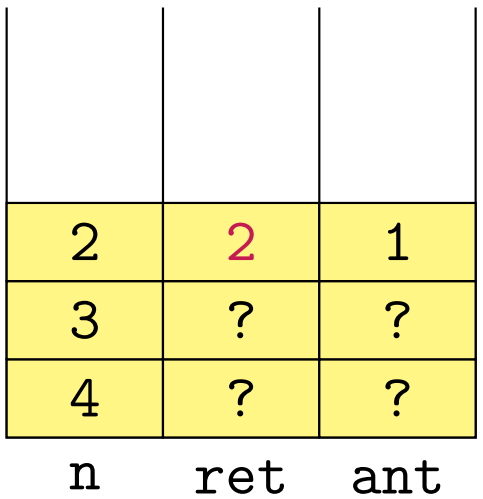
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

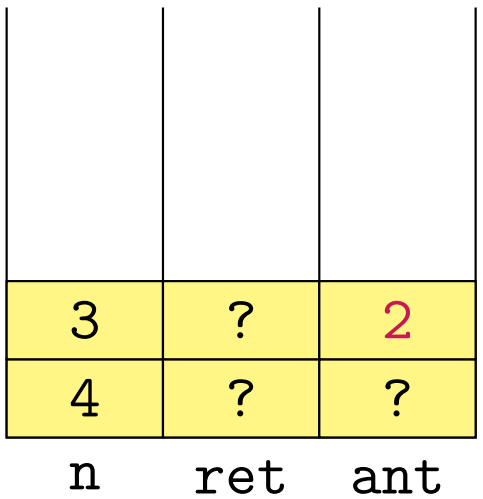
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

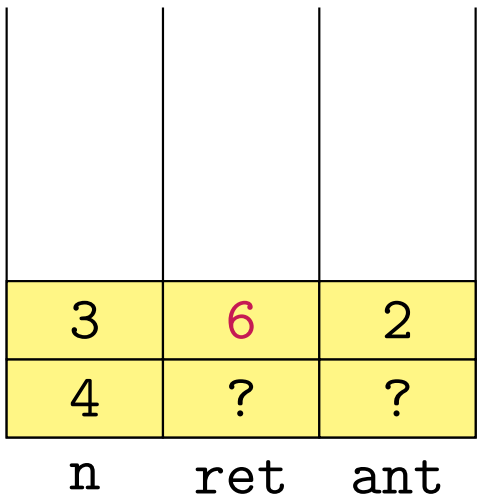
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

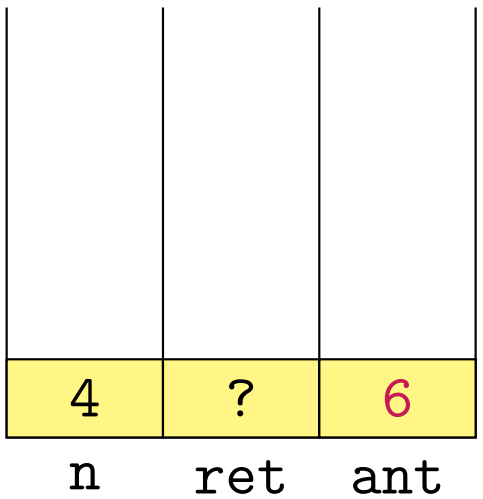
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

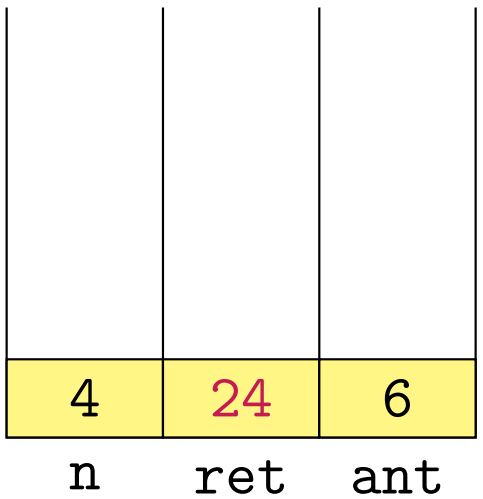
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

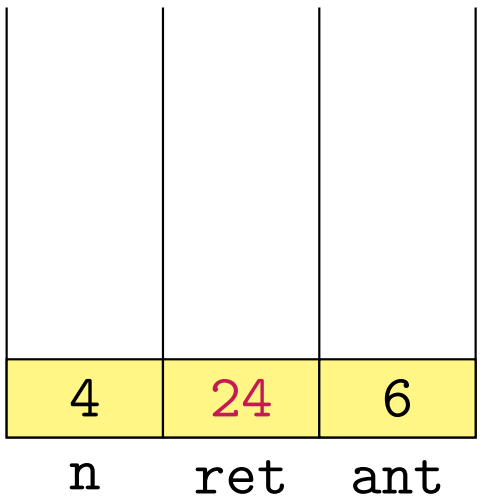
```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



Pilha de Chamadas

Estado da "pilha" de chamadas (*stack*) para `fat(4)`:

```
21 int fat(int n) {
22   int ret, ant;
23   if (n == 0)
24     ret = 1;
25   else {
26     ant = fat(n-1);
27     ret = n * ant;
28   }
29 return ret;
30 }
```



O *tamanho da stack* é limitado. Tente executar `fat(200000)`.

Pilhas e recursão

Quando empilhamos:

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

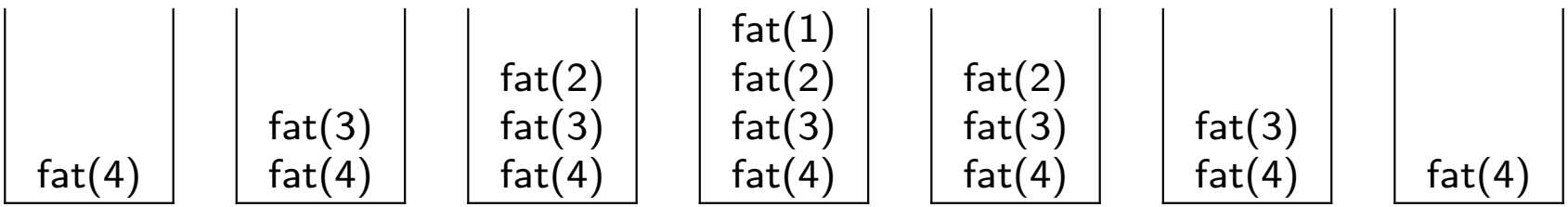
Quando desempilhamos:

- A chamada `fat(n)` retorna, apagamos o espaço para as variáveis locais
- Restabelecemos os valores das variáveis locais que tinham antes da chamada
- Precisamos também 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:

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



A *stack* (pilha de chamadas), é a pilha dos registros de ativação das chamadas em execução em um programa

Recursão geral

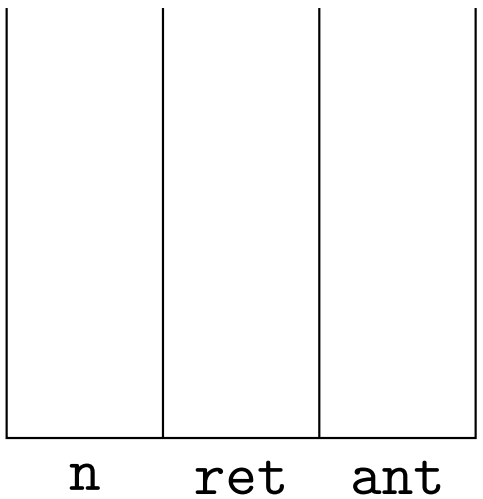
Eliminação de recursão:

- A partir de uma função recursiva sempre é possível escrever uma função equivalente sem recursão.
- A idéia geral é usar uma pilha para simular a stack e os registros de ativação.

Recursão geral

Versão de `fat` não recursiva (com uma pilha)

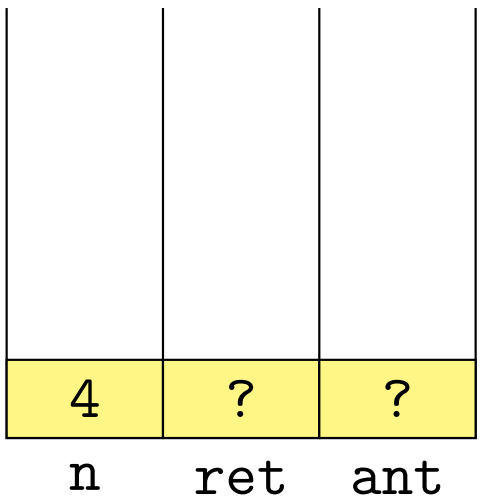
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

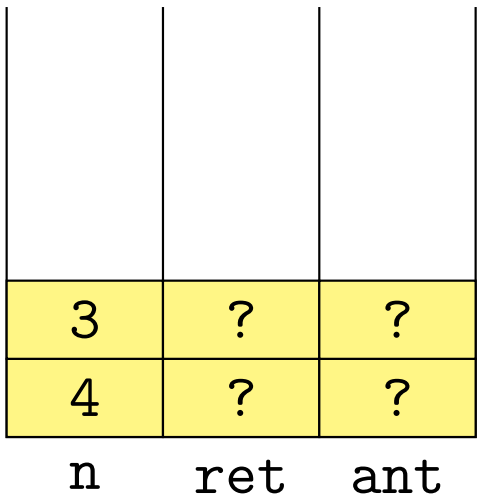
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

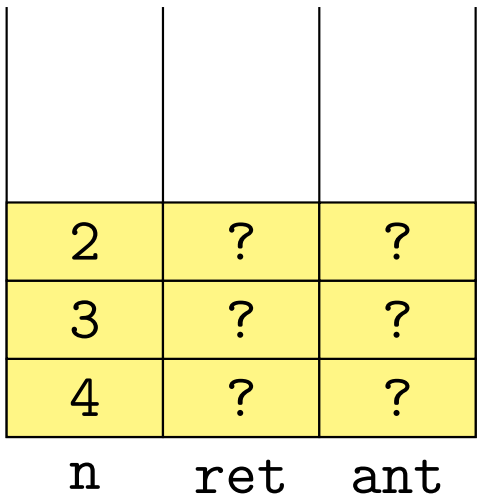
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

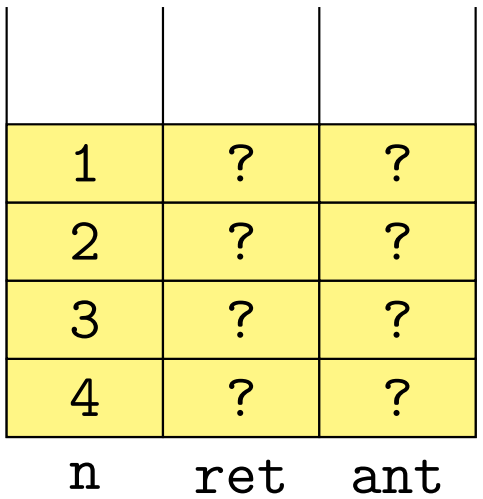
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```

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

Recursão geral

Versão de `fat` não recursiva (com uma pilha)

```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```

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

Recursão geral

Versão de `fat` não recursiva (com uma pilha)

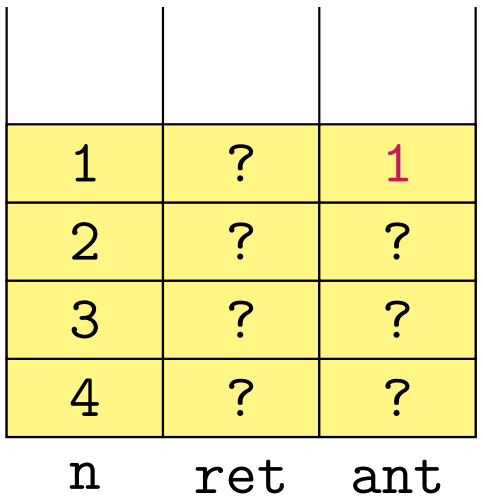
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```

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

Recursão geral

Versão de `fat` não recursiva (com uma pilha)

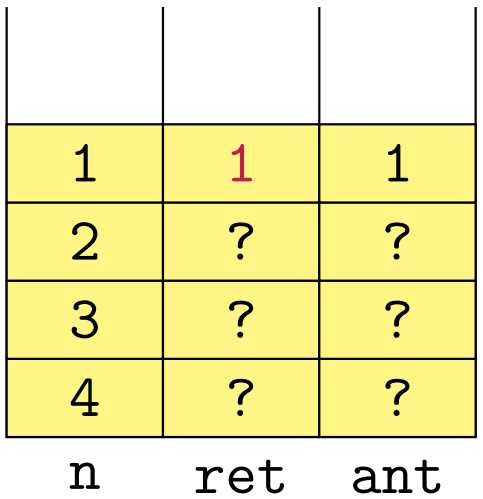
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

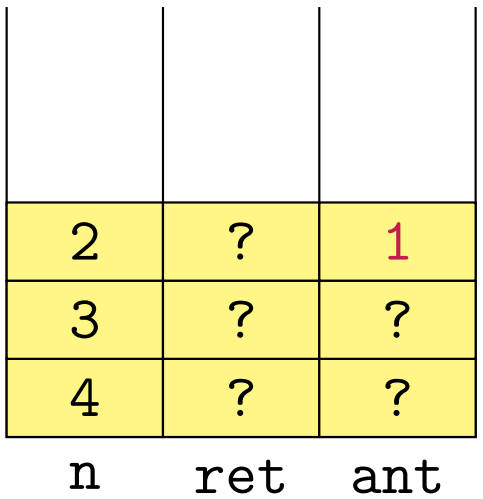
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

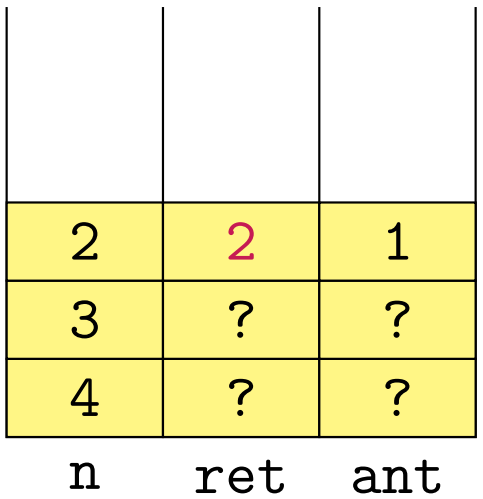
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

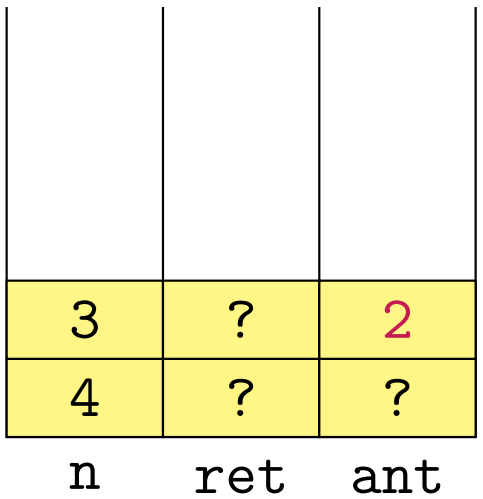
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

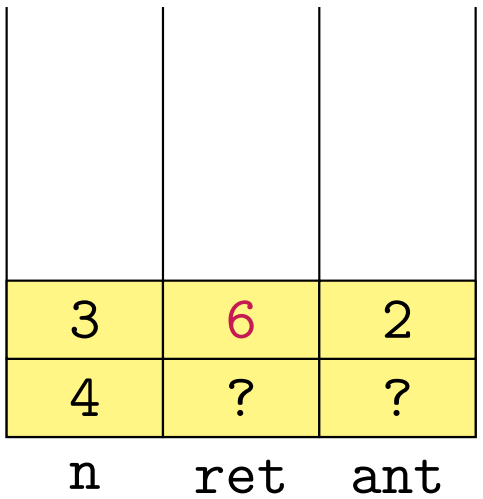
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

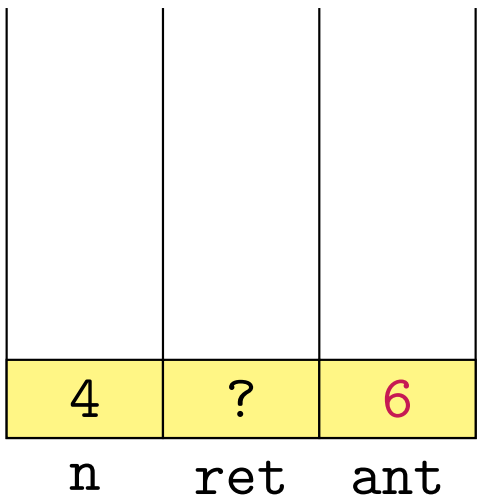
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

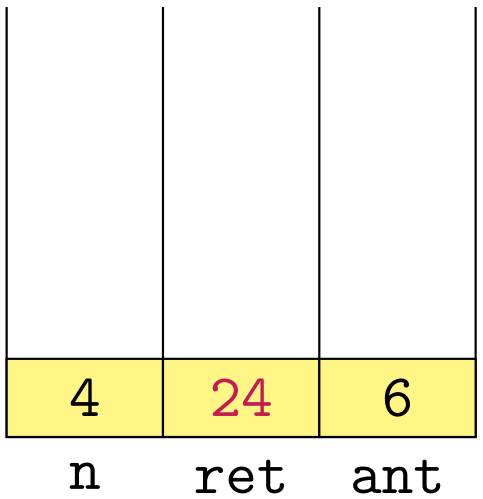
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

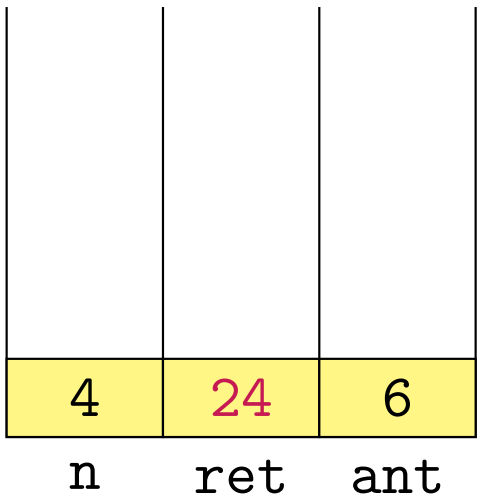
```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



Recursão geral

Versão de `fat` não recursiva (com uma pilha)

```
32 int fat_pilha(int n) {// n = 4
33   Pilha *P = pilha_criar();
34   while(n > 0){
35     pilha_adicionar(&P, n);
36     n--;
37   }
38   int ret=1, ant;
39   while(pilha_tamanho(P) > 0){
40     ant = ret;
41     n = pilha_topo(P);
42     pilha_remover(&P);
43     ret = ant * n;
44   }
45   pilha_destruir(&P);
46 return ret;
47 }
```



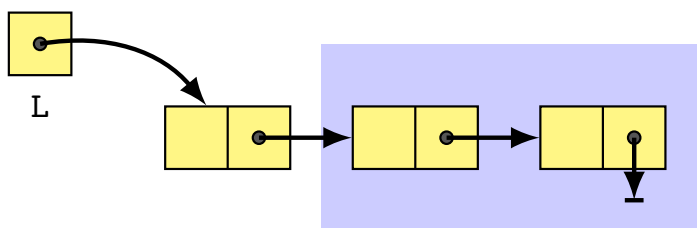
Tente executar `fat(200000)`.

Recursão de Cauda

Uma função em que a chamada recursiva é a última instrução é chamada de função recursiva em cauda.

Por exemplo:

```
1 int busca_rec(No *p, int v) {
2   if(p == NULL) return 0; // false
3   else if(p->dado == v) return 1;//true
4   else return busca_rec(p->prox, v);
5 }
```



Note que:

- exceto na base, o retorno não depende do valor das variáveis locais

Recursão de Cauda

Não precisamos de uma pilha para eliminar recursões de cauda!

```
1 int busca_rec(No *p, int v) {
2   if(p == NULL) return 0; // false
3   else if(p->dado == v) return 1;//true
4   else return busca_rec(p->prox, v);
5 }
```

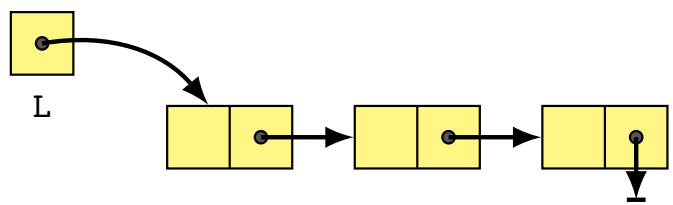
Recursão de Cauda

Não precisamos de uma pilha para eliminar recursões de cauda!

```
1  int busca_rec(No *p, int v) {
2    if(p == NULL) return 0; // false
3    else if(p->dado == v) return 1;//true
4    else return busca_rec(p->prox, v);
5  }
```

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

```
7  int busca_iterativa(No *p, int v) {
8    while(p != NULL || p->dado != v)
9      p = p->prox;
10   if(p == NULL) return 0; // false
11   else if(p->dado == v) return 1;//true
12 }
```



Recursão de Cauda

Alguns compiladores podem remover automaticamente recursões em cauda:

- Por exemplo, o gcc com a opção -O2:

```
1 gcc -O2 codigo.c -o codigo
```

Recursão de Cauda

Alguns compiladores podem remover automaticamente recursões em cauda:

- Por exemplo, o `gcc` com a opção `-O2`:

```
1 gcc -O2 codigo.c -o codigo
```

- Vamos ver como isso acontece em um exemplo simples:

`recursao_cauda.c`

```
3 int f(int x) {
4   printf("%d\n", x);
5   f(x+1);
6 }
```

Recursão de Cauda

Ao tentarmos executar `recursao_cauda`, temos um "estouro" da *stack*:

```
gcc recursao_cauda.c -o recursao_cauda
./recursao_cauda
```

```
...
...
261835
261836
Segmentation fault (core dumped)
```

Recursão de Cauda

Ao tentarmos executar `recursao_cauda`, temos um "estouro" da *stack*:

```
gcc recursao_cauda.c -o recursao_cauda
./recursao_cauda
```

```
...
...
261835
261836
Segmentation fault (core dumped)
```

- Vamos olhar o código `assembly` gerado:

```
gcc recursao_cauda.c -S
more recursao_cauda.s
```

---

http://www.delorie.com/djgpp/v2faq/faq8_20.html

Recursão de Cauda

recursao_cauda.s

```
37 main:
38 .LFB1:
39         .cfi_startproc
40         pushq        %rbp
41         .cfi_def_cfa_offset 16
42         .cfi_offset 6, -16
43         movq         %rsp, %rbp
44         .cfi_def_cfa_register 6
45         movl         $1, %edi
46         call         f
47         movl         $0, %eax
48         popq         %rbp
49         .cfi_def_cfa 7, 8
50         ret
51         .cfi_endproc
```

```
 9 f:
10 .LFB0:
11         .cfi_startproc
12         pushq        %rbp
13         .cfi_def_cfa_offset 16
14         .cfi_offset 6, -16
15         movq         %rsp, %rbp
16         .cfi_def_cfa_register 6
17         subq         $16, %rsp
18         movl         %edi, -4(%rbp)
19         movl         -4(%rbp), %eax
20         movl         %eax, %esi
21         movl         $.LC0, %edi
22         movl         $0, %eax
23         call         printf
24         movl         -4(%rbp), %eax
25         addl         $1, %eax
26         movl         %eax, %edi
27         call         f
28         nop
29         leave
30         .cfi_def_cfa 7, 8
31         ret
32         .cfi_endproc
```

Podemos ver o registro de ativação no preambulo de cada função.

Recursão de Cauda

Ao compilar `recursao_cauda.c` com a opção `-O2`, o `gcc` identifica e remove a recursão em cauda:

```
gcc -O2 recursao_cauda.c -o recursao_cauda
./recursao_cauda
```

```
...
...
261835
261836
...
...
```

Recursão de Cauda

Ao compilar `recursao_cauda.c` com a opção `-O2`, o `gcc` identifica e remove a recursão em cauda:

```
1 gcc -O2 recursao_cauda.c -o recursao_cauda
2 ./recursao_cauda
```

```
1 ...
2 ...
3 261835
4 261836
5 ...
6 ...
```

- Por último, vamos olhar o código `assembly` gerado:

```
1 gcc -O2 recursao_cauda.c -S
2 more recursao_cauda.s
```

Recursão de Cauda

recursao_cauda.s

```
33 main:
34 .LFB12:
35         .cfi_startproc
36         subq        $8, %rsp
37         .cfi_def_cfa_offset 16
38         movl        $1, %edi
39         call        f
40         .cfi_endproc
```

```
10 f:
11 .LFB11:
12         .cfi_startproc
13         pushq       %rbx
14         .cfi_def_cfa_offset 16
15         .cfi_offset 3, -16
16         movl        %edi, %ebx
17         .p2align 4,,10
18         .p2align 3
19 .L2:
20         movl        %ebx, %esi
21         movl        $.LC0, %edi
22         xorl        %eax, %eax
23         addl        $1, %ebx
24         call        printf
25         jmp         .L2
26         .cfi_endproc
```

A chamada recursiva foi substituída por um laço.

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

Fim

Dúvidas?

Roteiro

Referências

1. Materiais adaptados dos slides do Prof. Rafael C. S. Schouery, da Universidade Estadual de Campinas.
2. Feofiloff, Paulo. Algoritmos em linguagem C. Elsevier Brasil, 2009.