
Listas Genéricas (não homogêneas) e Listas Generalizadas

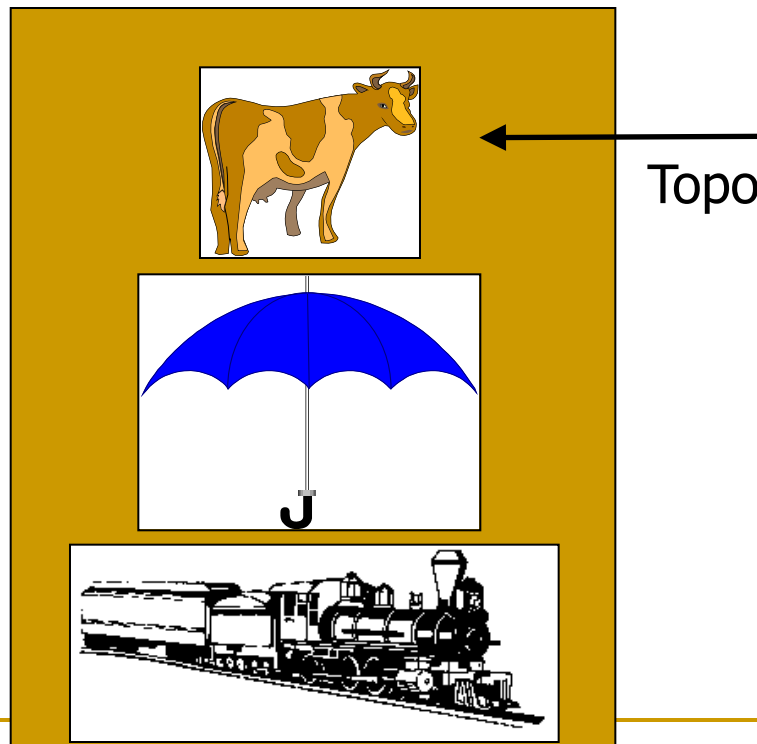
14/10/2010

Lista (Pilha e Fila) genérica

- Possibilidade de usar uma mesma estrutura para armazenar informações diferentes
 - Integer, char, record, etc.
 - Até agora para usarmos Pilhas, Filas e Listas com tipos de dados diferentes
 - atualizávamos o arquivo “elemento.h” mas sempre escolhíamos um único elemento (int ou float ou char) → **lista homogênea**
-

Lista genérica, não homogênea

- Como inserir uma vaca, um guarda-chuva e um trem em uma mesma pilha?



Lista genérica

■ Solução 1

- Definem-se vários campos de informação
- Usam-se somente os necessários

```
struct no {  
    char info1;  
    int info2;  
    struct no *prox;  
}
```

- Desvantagem: memória alocada desnecessariamente
 - Alternativa?

Lista genérica

■ Solução 2

- Definem-se vários ponteiros
- Aloca-se memória conforme necessidade

```
struct no {  
    char *info1;  
    int *info2;  
    struct no *prox;  
}
```

Para registros grandes pode ser uma solução, mas para tipos simples não, dado que ponteiro ocupa 4 bytes no mínimo, **alocado desnecessariamente** quando não usados

Lista genérica

- Solução 3: registro variante
- Em C: Union

```
struct no {  
    union {  
        int ival;  
        float fval;  
        char cval;  
    } elemento;  
    struct no *next;  
}
```

Union permite que uma variável seja interpretada de diferentes formas.

A memória do maior elemento (no caso acima o float) é alocada e o usuário deve cuidar do bom uso dela.

Pode-se guardar inclusive uma **tag** dentro de cada nó:

```
struct no {  
    int tipo; // 1,2,3  
    union {  
        int ival;  
        float fval;  
        char cval;  
    } elemento;  
    struct no *next;  
}
```

```
struct no *p;  
...  
p->tipo = 1; /*inteiro*/  
p->elemento.ival = 256;  
...  
p->tipo = 3; /*char*/  
p->elemento.cval = 'n';  
...
```

Aplicação de lista genérica: Tabela de Símbolos de um compilador para linguagens orientadas a blocos

TS implementada estaticamente como uma “pilha”

```
Type categoria = (constante, tipo, variavel, procedimento, funcao, parametro);
classset = (valor, referencia, procedimento, funcao);
dim = record
    inf, sup: integer
end;
item = record
    ident: string[tam_max];
    nivel: integer;
    case categ: categoria of
        constante: (case tipoc: integer of
            1: (valori: integer);
            2: (valorc: char);
            3: (valorr: real);
            4: (valors: string);
            5: (valorb: boolean););
        tipo: (nbytes: integer; dimensao: dim; tipo_elem: integer);
        procedimento: (npar1: integer; end1: integer);
        funcao: (npar2: integer; end2: integer; tipo_f: integer);
        parametro: (classe: classset; end3: integer; tipo_p: integer);
        variavel: (end4: integer; tipo_v: integer)
    end;
TS: record
    pilha: array [1..max] of item;
    topo: integer
end;
```

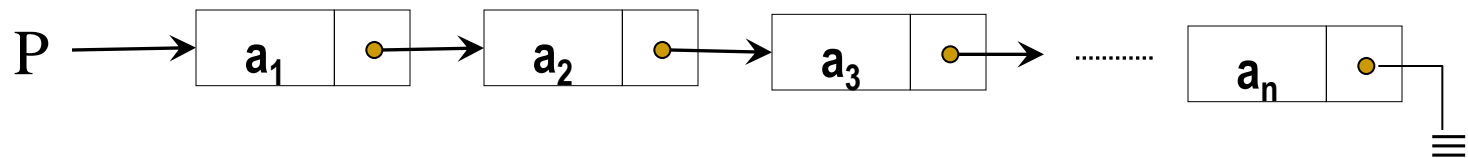
Cada um dos 6 tipos diferentes de Identificadores exige atributos variados. Em Pascal a union é chamada de record variante (possui um case).

Por exemplo, para uma **variável** precisamos saber de seu endereço de memória, e seu tipo, além das informações gerais como nome e escopo.

Generalizando o conceito de Lista

- Uma lista (a_1, a_2, \dots, a_n) pode ser definida como uma seqüência constituída do elemento a_1 seguido da lista (a_2, \dots, a_n) que é definida recursivamente, de forma análoga, até que a lista (a_n) seja formada por a_n seguido da lista vazia $()$

Implementação dinâmica reflete essa situação:



(P e o elemento apontado por $P^{\wedge}.lig$ são do mesmo tipo)

Generalizando o conceito de Lista

- Até agora, consideramos todos os a_i do mesmo tipo, e sempre um **átomo** (elemento indivisível, não lista)
- Podemos considerar que cada elemento a_i da lista poderia também ser uma lista (chamada **sub-lista**)

Ex.: $L = (a, (b, c), d, (e), ())$

a_1 a_2 a_3 a_4 a_5

L tem 5 elementos

- a_2 , a_4 e a_5 são **sub-listas**
- a_1 e a_3 são **átomos**

Generalizando o conceito de Lista

- Lista típica em **LISP**:

$L1 = (a, (b, c))$

- Lista típica em **Prolog**:

$L2 = [a, [b, c]]$

- Em Prolog e Lisp Listas são **nativas** e, portanto, implementadas com eficiência.
- Ambas as listas contêm dois elementos
 - primeiro elemento é o **átomo** a;
 - segundo elemento é a sub-lista formada pelos elementos b e c.
 - Como **representar** essas listas?

Listas Generalizadas

Definição

Uma lista generalizada A é uma seqüência finita de $n \geq 0$ elementos $\alpha_1, \alpha_1, \dots, \alpha_n$ em que α_i são **átomos** ou **listas**. Os elementos α_i , $0 \leq i \leq n$, que não são átomos, são chamados **sub-listas** de A .

Listas Generalizadas

- Elemento pode ser um átomo ou uma outra lista (sub-lista)
- Elemento pode ser **representado** pela seguinte estrutura de nó



- CABEÇA (**CAR**): um átomo ou um ponteiro para uma outra lista
- CAUDA (**CDR**) ligação para a cauda da lista ('próximo elemento')
- **TAG** é 0 (CABEÇA é átomo) ou é 1 (CABEÇA é ponteiro)

Listas Generalizadas

Dada uma lista generalizada

$$A = [\alpha_1, \alpha_2, \dots, \alpha_n]$$

Se $n \geq 1$, então:

α_1 é a cabeça (car) de A,
 $(\alpha_2, \dots, \alpha_n)$ é a cauda (cdr) de A

Exemplos:

1. $L1 = ()$: tem tamanho 0 e contém a lista vazia
2. $L2 = (())$: tem tamanho 1 e contém a lista nula
3. $L3 = (a)$: tem tamanho 1 e contém o átomo a
4. $L4 = (a,(b,c))$: tem tamanho 2 e contém o átomo a e a lista (b,c)

$$\text{car}(L4) = a, \text{cdr}(L4) = ((b,c))$$

(note que cdr é sempre uma lista, já car pode ser átomo ou lista)

Implementação

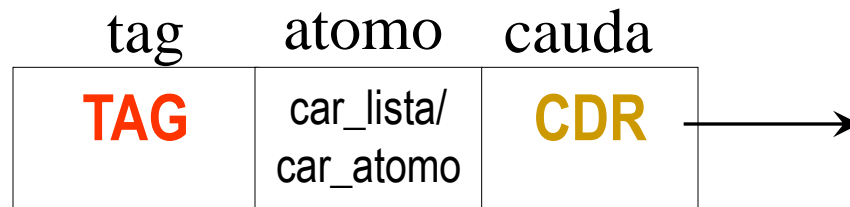
```
struct no {  
    int tag; // 1,0  
    union {  
        struct no *car_lista;  
        tipo_elem car_atomo;  
    } elemento;  
    struct no *cdr;  
}  
typedef struct no *nopr;
```

Lista_Gen.c

Lista_Gen.h

nopr lista;

pont ou

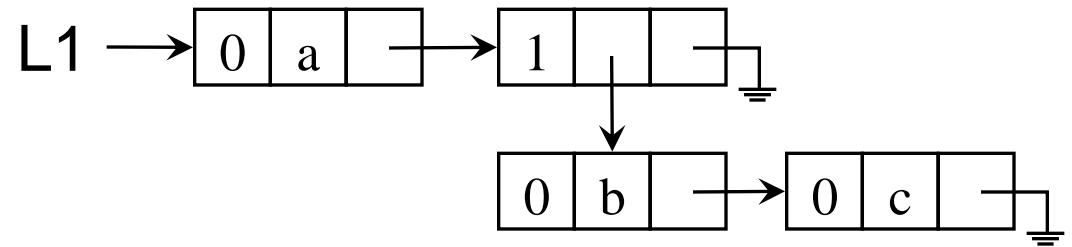


Vantagens de Listas Generalizadas

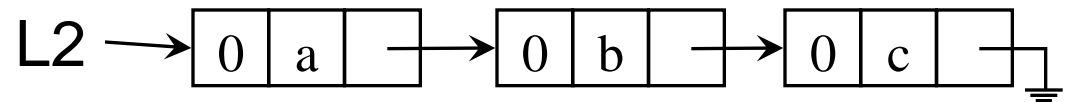
- Multifuncional: com ela podemos armazenar informações das mais simples às mais complexas.
 - Além da sua importância como opção de armazenamento de dados, a lista generalizada é uma estrutura de dados recursiva, e nos permitirá **por em prática a implementação de funções recursivas** que iniciamos na aula sobre listas ordenadas.
-

Mais Exemplos com Representação

$L1 = (a, (b, c))$



$L2 = (a, b, c)$



Cabeça(L2)? Cauda(L2)? Cabeça(Cauda(L2))?

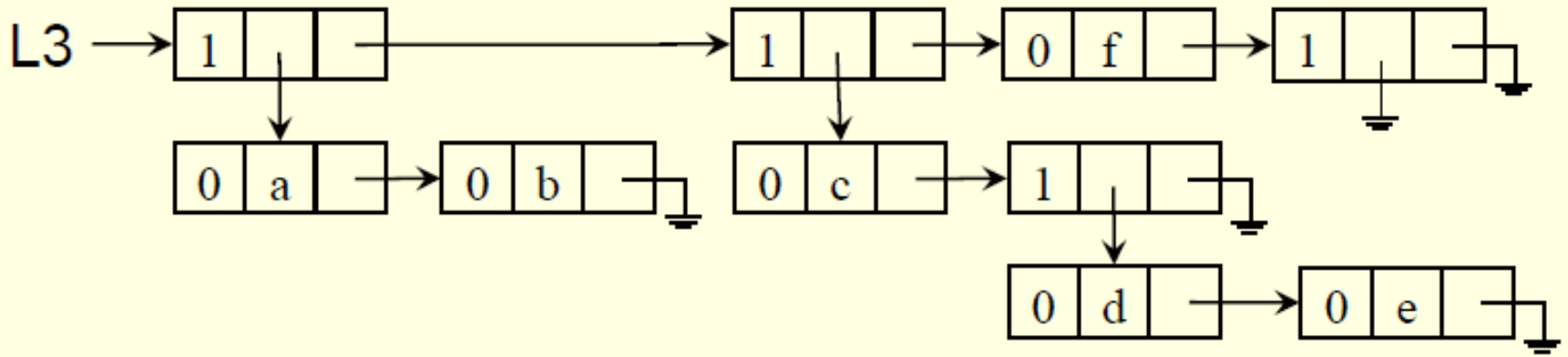
Cabeça(L1)? Cauda(L1)? Cabeça(Cauda(L1))?

Lista generalizada: exercício

Façam a representação da lista L3 ((a,b),(c,(d,e)),f,())



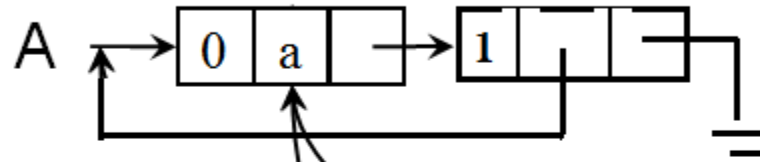
Solução



Variações de Listas Generalizadas

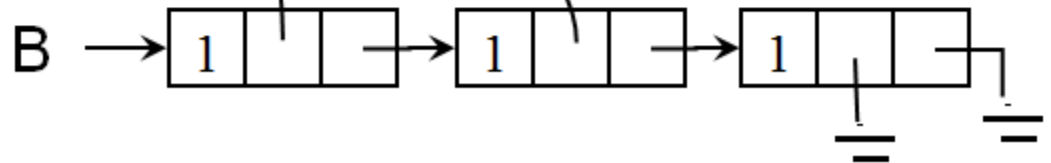
Listas Recursivas

$A = (a, A)$



Listas Compartilhadas

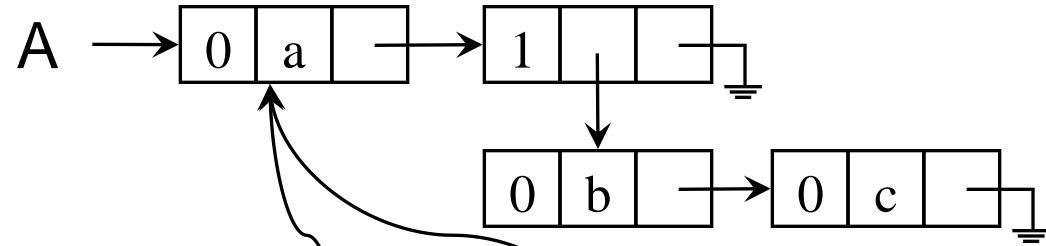
$B = (A, A, ())$



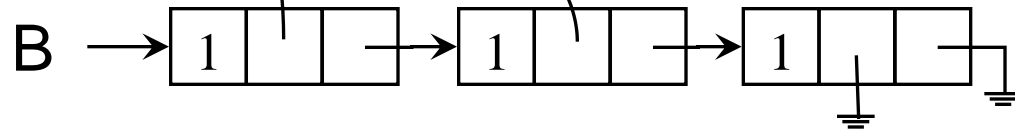
Compartilhamento pode resultar em grande economia de memória. Entretanto, esse tipo de estrutura cria problemas quando desejamos **eliminar** ou **inserir** nós na frente da lista.

Variações de Listas Generalizadas

$A = (a, (b, c))$



$B = (A, A, ())$

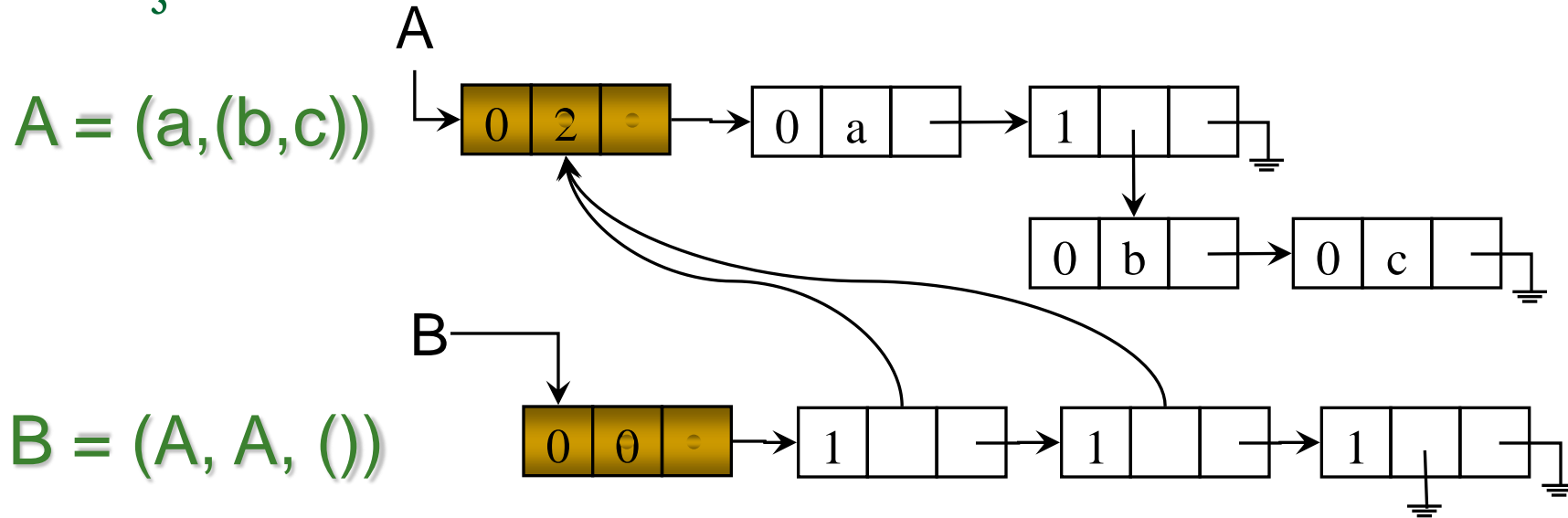


Ex. o que acontece quando o primeiro elemento de A é eliminado? Ou se for inserido um novo elemento como o primeiro da lista A?

Note que, em geral, não se sabe quantos ponteiros estão apontando para a estrutura, e tampouco de onde eles vêm!

Ainda que essa informação fosse conhecida, a manutenção seria custosa.

Variações de Listas Generalizadas



Solução: uso de nó cabeça (*head*).

Pode aproveitar esse nó para manter (no campo CAR) um contador dos nós que apontam para a lista.

Contador de referências: o número de ponteiros (variáveis de programa ou ponteiros a partir de outras listas) para aquela lista. Quando o contador é zero, os nós podem ser retornados à memória disponível.

Operações



```
noptr P,Q;
```

Para atribuir:

```
P->tag = 0;
```

```
P->elemento.car_atomo =  
info;
```

ou

```
P->tag = 1;
```

```
P->elemento.car_lista = Q;
```

```
struct no {  
    int tag; // 1,0  
    union {  
        tipo_elem car_atomo;  
        struct no *car_lista;  
    } elemento;  
    struct no *cdr;  
}
```

```
typedef struct no *noptr;  
noptr lista;
```

```
Noptr P,Q;
```

Para usar/acessar:

```
if (P->tag)
```

```
P->elemento.car_lista...
```

```
else
```

```
P->elemento.car_atomo...
```

Operações Úteis

`void Cria (noptr *L);`

`// Cria uma lista vazia`

`void Concatena (noptr *L1, noptr L2);`

`// Insere L2 no final de L1. Se L1= (a,(b,c)) e L2=(d) → L1=(a,(b,c),d)`

`noptr BuscaAtomo_Rec(noptr L, tipo_elem x)`

`// Busca ocorrência do átomo x na lista generalizada L; retorna seu endereço. Devolve null se não achou`

`int Prof(noptr S)`

`// Calcula a profundidade da lista generalizada S`

`noptr Copia(noptr L)`

`// cria uma copia da lista generalizada L`

`int Igual(noptr S, noptr T)`

`// Verifica a igualdade de duas listas generalizadas S e T`

`void Lista(noptr L)`

`// Imprime a lista L`

`void Esvazia(noptr *L)`

`//Esvazia uma lista, liberando a sua memória alocada`

... Entre outras...

Façam as operações:

// Cria uma Lista Generalizada vazia. Deve ser usado antes das outras operações do TAD.

void Cria (noptr * L);

// insere L2 no final de L1. Se $L1 = (a, (b, c))$ e $L2 = (d) \rightarrow L1 = (a, (b, c), d)$.

// Caminha na lista principal

void Concatena (noptr *L1, noptr L2);

// Imprime a lista generalizada L.

void Lista(noptr L);

//Esvazia uma lista, liberando a sua memória alocada.

void Esvazia(noptr *L);

Operações

```
void Cria(noptr *L) {  
    *L = NULL;  
}
```

```
void Concatena(noptr *L1, noptr L2) {  
    noptr p;  
  
    if (*L1==NULL) *L1 = L2;  
    else {  
        p = *L1;  
        while (p->cdr != NULL) p = p->cdr;  
        p->cdr = L2;  
    }  
}
```

Operações

```
void Lista(noptr L) {  
  
    if (L != NULL)  
        if (L->tag == 0) {  
            printf("%c ", L->elemento.car_atomo);  
            Lista(L->cdr);  
        }  
        else {  
            printf(" (");  
            Lista(L->elemento.car_lista);  
            printf(") ");  
            Lista(L->cdr);  
        }  
}
```

OBS: esta função deve chamar uma outra (definida em Lista_Gen.h) para imprimir o tipo de dado usado, que também deve ser definido em Lista_Gen.h

Operações

```
void Esvazia(noptr *L){
```

```
  noptr t;
```

```
  if ((*L) != NULL) {
```

```
    if ((*L)->tag == 1) {
```

```
      Esvazia(&(*L)->elemento.car_lista);
```

```
      t = *L; *L = (*L)->cdr; free(t);
```

```
      Esvazia(&(*L));
```

```
    }
```

```
  } else {
```

```
    t = *L; *L = (*L)->cdr; free(t);
```

```
    Esvazia(&(*L));
```

```
  }
```

```
}
```

```
}
```

Operações de Inserção de Elementos

// Insere o primeiro atomo numa Lista generalizada.

```
void Insere_Prim_Atomo(noptr *L, tipo_elem valor);
```

// Insere a primeira lista numa Lista generalizada.

```
void Insere_Prim_Lista(noptr *L, noptr L1);
```

// Insere um atomo no início de uma Lista generalizada.

```
void Insere_Inicio_Atomo(noptr *L, tipo_elem valor);
```

// Insere uma lista no início de uma Lista generalizada.

```
void Insere_Inicio_Lista(noptr *L, noptr L1);
```

Exercícios

- Implementar uma função para **buscar** um átomo x numa lista generalizada, e devolver seu endereço:
 - (1) considere apenas a **lista principal** (versão não recursiva);
 - (2) considere que **x pode estar em qualquer sublista** (versão recursiva)
- Implementar uma função para verificar se duas listas generalizadas são **iguais (estrutura e conteúdo)**
 - Façam a versão recursiva

Busca na lista principal

{ busca ocorrência do átomo x na lista generalizada L; retorna seu endereço. Devolve null se não achou }

```
noptr BuscaAtomo_Lista_P(noptr L, tipo_elem x) {  
int achou=0;  
while ((L!=NULL) && (!achou)) {  
    if ((L->tag==0) && (L->elemento.car_atomo==x))  
        achou=1;  
    else L=L->cdr;  
}  
return L;  
}
```

Busca recursiva

{ busca ocorrência do átomo x na lista generalizada L; retorna seu endereço. Devolve null se não achou }

```
noptr buscaAtomo(noptr L, tipo_elem x)
```

```
noptr buscaAtomo(noptr L, tipo_elem x)
```

```
// Só busca se lista não for vazia
```

```
// Verificar se o nó é átomo ou lista. Se for atomo compara, se achou pára a busca. Se for lista procura recursivamente.
```

```
// se não achou no nó busca na cauda recursivamente.
```

```
noptr BuscaAtomo_Rec(noptr L, tipo_elem x){  
  
if (L==NULL)  
    return NULL;  
else if (L->tag==0) { // se achou na cabeça, retorna endereço  
    if (L->elemento.car_atomo==x)  
        return L;  
    else return BuscaAtomo_Rec(L->cdr,x); // senão busca na cauda  
}  
else if (L->tag==1) { // se não achou na lista, procura na cauda  
    if (BuscaAtomo_Rec(L->elemento.car_lista,x) == NULL)  
        return BuscaAtomo_Rec(L->cdr,x);  
}  
}
```


int Igual(noptr S, noptr T)

// Verifica a igualdade (de estrutura e conteúdo) de duas listas generalizadas S e T. Retorna 1 (true) se são e 0 (false) se não são iguais

Se S e T são vazias **então** resp:= true // caso 1: e & c iguais

Senão **se** S e T são não vazias **então** // caso 2: e igual

Início

se car(S) e car(T) são de igual tipo // tipo igual de cabeça: e igual

então // testa conteúdo

Início

se ambos são átomos **então**

resp:= (testa se os átomos são iguais) // caso 2.1/2.3: c igual/dif

senão resp:= (testa se car(S) = car(T)) // caso 2.2/2.3: c igual/dif

se resp **então** (testa se cdr(S) = cdr(T)) // testa cauda se c acima é igual

Fim

senão resp := false // caso 2.3: tipo diferente de cabeça

Fim

senão resp:=false // caso 3: e diferente

// Verifica a igualdade (de estrutura e conteúdo) de duas listas generalizadas S e T. Retorna 1 (true) se são e 0 (false) se não são iguais

```
int Igual(noptr S, noptr T){
    if ((S==NULL) && (T==NULL)) // caso 1: e & c iguais
        return 1;
    else if ((S==NULL) || (T==NULL)) // caso 3: e diferente
        return 0;
    else // caso 2: e igual (checar tipo e conteúdo da cabeça)
        if (((S->tag== 0) && (T->tag==0)) && // 2.1 cabeça = átomos
            (S->elemento.car_atomo==T->elemento.car_atomo)) // 2.1 iguais
            return Igual(S->cdr,T->cdr); // checa cauda da lista
        else if (((S->tag== 1) && (T->tag==1)) && // 2.2 cabeça = listas
            (Igual(S->elemento.car_lista,T->elemento.car_lista))) // 2.2 iguais
            return Igual(S->cdr,T->cdr); // checa cauda da lista
        else
            return 0; // 2.3 c diferente ou tipo diferente
}
```