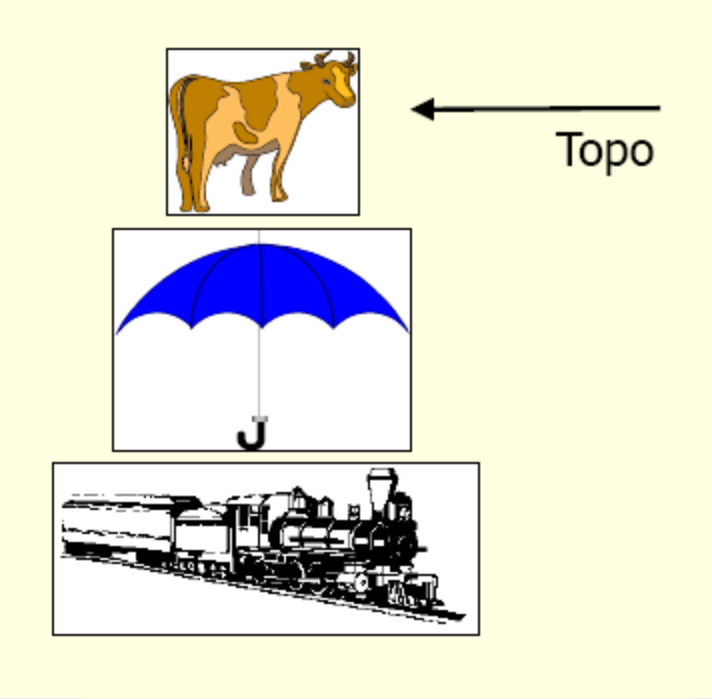
Lista não homogênea

Lista não homogênea

- Lista "genérica"
  - Possibilidade de usar uma mesma estrutura para armazenar informações diferentes
    - Inteiro, caracter, estrutura, etc.
  - Não é necessário definir blocos de memória diferentes

Lista não homogênea

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



Lista não homogênea

- Solução 1
  - Definir vários campos de informação
  - Usar somente os necessários

```
struct NO {
      char info1;
      int info2;
      struct NO *proximo;
};
```

- Desvantagem: memória alocada desnecessariamente

Lista não homogênea

- Solução 2
  - Definir vários ponteiros
  - Alocar memória conforme necessidade

```
struct NO {
      char *info1;
      int *info2;
      struct NO *proximo;
};
```

- Desvantagem: memória (dos ponteiros não usados) alocada desnecessariamente

Lista não homogênea

- Solução 3
  - Usa-se um registro/estrutura variante

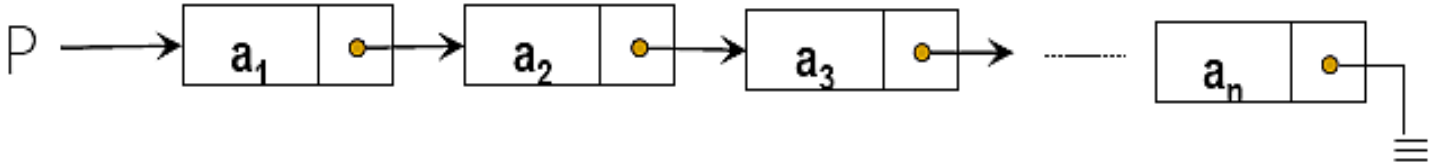
```
struct NO {
      union{
            int ival;
            float fval;
            char cval;
       }elemento;
       int tipo;
       struct NO *prox;
}
```

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

Lista generalizada

Conceito de Lista

- Uma lista $a_1, a_{2..,} a_n$ pode ser definida como uma sequência constituída do elemento a_1 seguido da lista $a_2, a_{3..,} a_n$ que é definida recursivamente, de forma análoga, até que a lista (a_n) seja formada por a_n seguido da lista vazia (NULL)
- Implementação dinâmica reflete essa situação:



- (P e o elemento apontado por P^.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 de forma eficiente.

- 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?

Lista Generalizada

- Definição formal
  - Uma lista generalizada A é uma sequência finita de $n \geq 0$ elementos $\alpha_1, \alpha_1, .., \alpha_n$, em que α_i são átomos ou listas. Os elementos α_i, com $0 \leq i \leq n$ que não são átomos são chamados sublistas de A.
- Estrutura básica do bloco de memória

Tipo	Cabeça	Cauda

Lista Generalizada

- Uma lista generalizada é aquela que pode ter como elemento ou um átomo ou uma outra lista (sub-lista)
  - Átomo: integer, real, char, string, etc.

Tipo	Cabeça	Cauda

- CABEÇA : um átomo ou um ponteiro para uma outra lista
- CAUDA: ligação para a cauda da lista (próximo elemento)
- TIPO é 0 (CABEÇA é átomo) ou é 1 (CABEÇA é ponteiro)

Lista Generalizada

- Dada uma lista generalizada

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

Se $n \geq 1$, então:

α_1 é a cabeça (car) de A,

$(\alpha_2, \ldots \alpha_n)$ é a cauda (cdr) de A

cabeça

cauda

- 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)

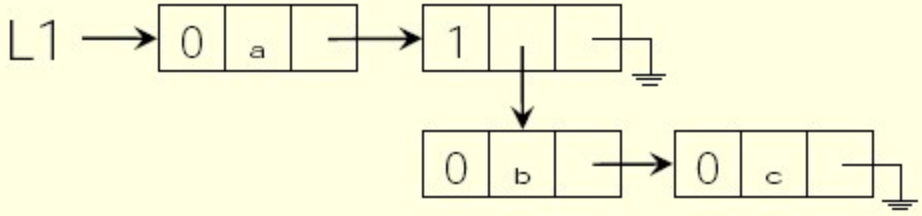
cabeça (L4) = a, cauda(L4) = ((b,c))

- (note que cabeça é sempre uma lista, já cauda pode ser átomo ou lista)

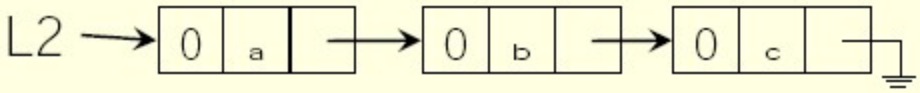
Lista Generalizada

- Exemplos de representação

L1 = (a,(b,c))



L2 = (a,b,c)

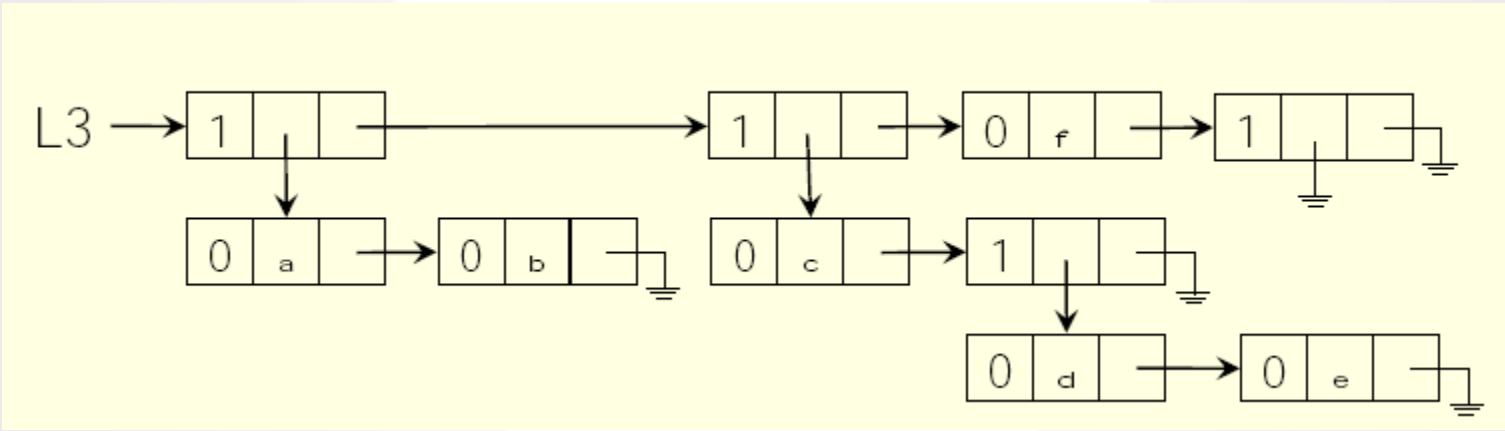


Lista Generalizada

- Exercício: faça a representação da lista L3 ((a,b),(c, (d,e)),f,())

Lista Generalizada

- Exercício: faça a representação da lista L3 ((a,b),(c, (d,e)),f,())



Vantagens de Listas Generalizadas

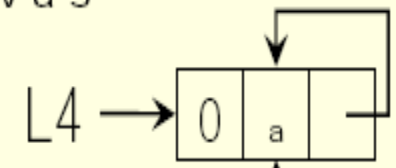
- Multifuncional: com ela podemos armazenar informações das mais simples às mais complexas.

- 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.

Variações de Lista Generalizada

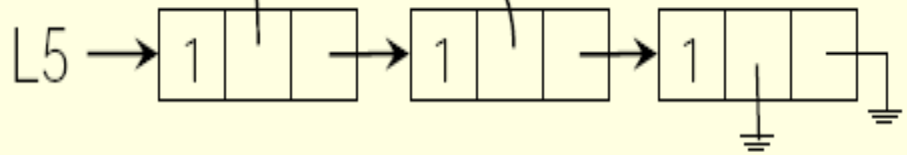
Listas Recursivas

L4 = (a,L4)



Listas Compartilhadas

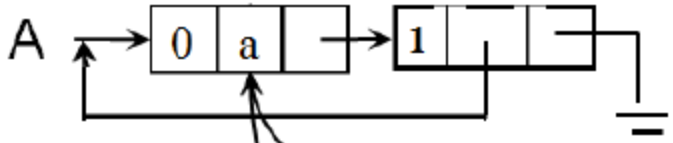
L5 = (L4,L4,())



Variações de Lista Generalizada

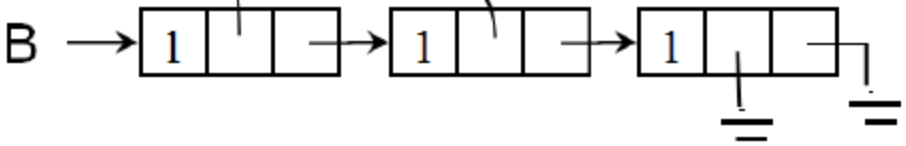
Listas Recursivas

A = (a, A)



Listas Compartilhadas

B = (A, A, ())

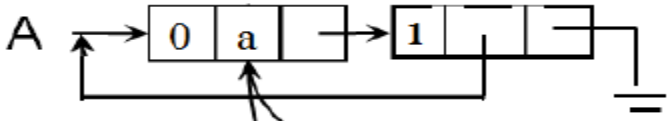


- Compartilhamento pode resultar em grande economia de memória. Entretanto, esse tipo de estrutura cria problemas quando desejamos eliminar ou inserir nós na frente da lista.

Variações de Lista Generalizada

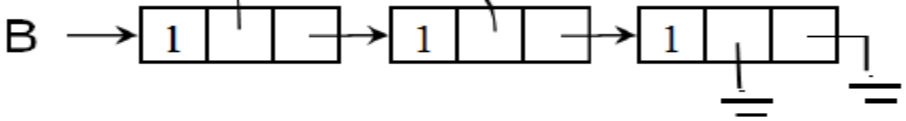
Listas Recursivas

A = (a, A)



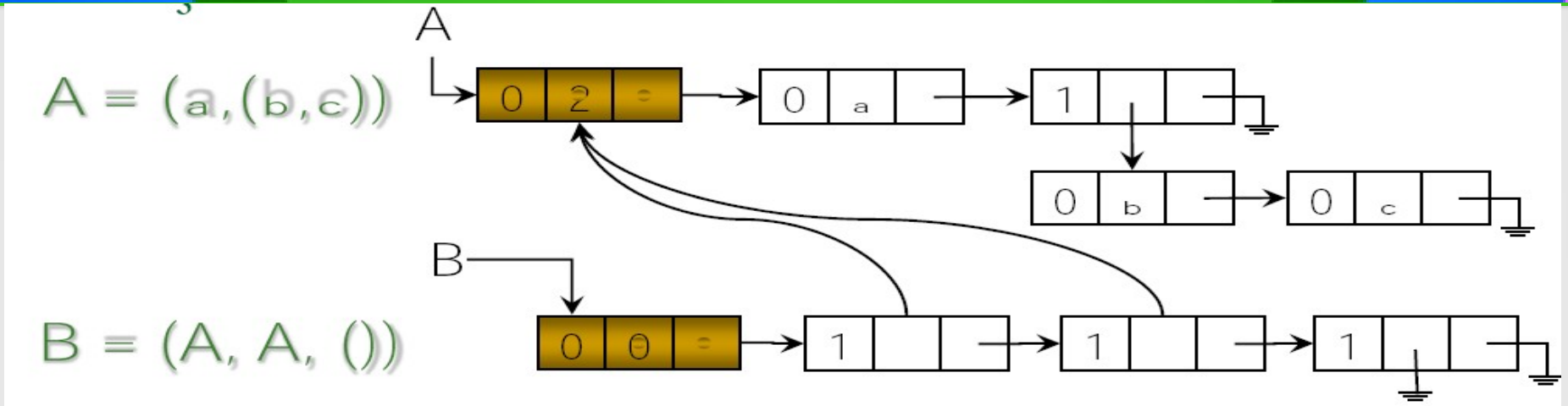
Listas Compartilhadas

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 Lista Generalizada

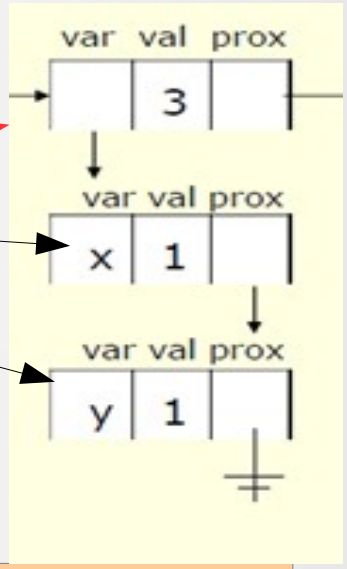


- Solução: uso de nó cabeça (head).
- Pode aproveitar esse nó para manter (no campo cabeça) um contador dos nós que apontam para a lista.
- Contador de referências: o número de ponteiros (variáveis de programa ou ponteiros a partir de outras listas) para aquela lista. Quando o contador é zero, os nós podem ser retornados à memória disponível.

Lista Generalizada

- Declaração
  - Union

```
typeder struct bloco {
     union{
          char atomo;
          struct bloco *sublista;
          char cval;
       }info;
       int tipo;
       struct bloco *prox;
} tNO;
```



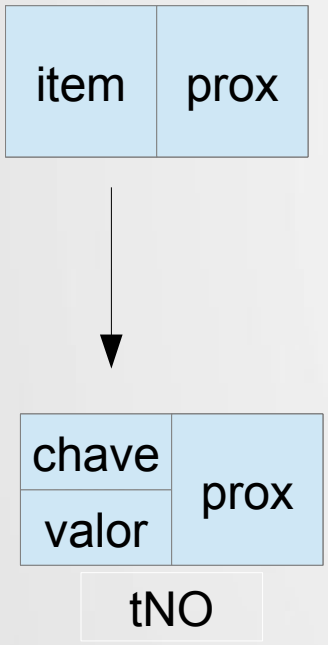
- Union permite que uma variável seja interpretada de diferentes formas.
- A memória do maior elemento (no caso acima o float) é alocada e o usuário deve cuidar do bom uso dela

Relembrando: Lista Ligada

```
#define MAX 10

typedef struct {
  int chave;
  int valor;
} ITEM;
//representa a lista de itens

typedef struct NO {
    ITEM item;
    struct NO *proximo;
} tNO;
```

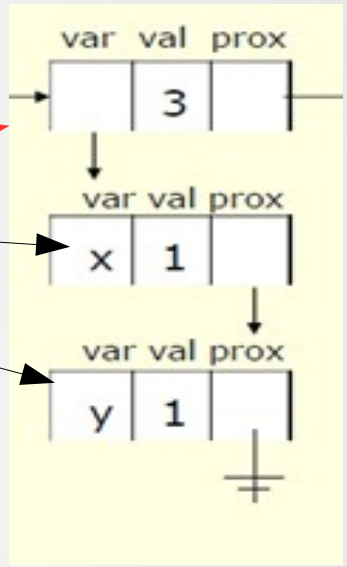



```
typedef struct {
   tNO *inicio;
   tNO *fim;
} LISTA_Ligada;
```

Lista Generalizada

ITEM atomo;

```
typedef struct bloco {
      union{
            char atomo;
            struct bloco *sublista;
            char cval;
       }info;
       int tipo;
       struct bloco *prox;
} tNO;
```




```
typedef struct {
   tNO *noptr;
} Lista_Gen;
```

```
typedef struct {
  int chave;
  int valor;
} ITEM;
//representa a lista de itens
```

tipo	info	prox

Operações úteis

- void Cria (Lista_Gen *L); // Cria uma lista vazia
- void Concatena (Lista_Gen *L1, Gen *L2);
- // Insere L2 no final de L1. Se L1= (a,(b,c)) e L2=(d) L1=(a, (b,c),d)
- *tNO BuscaAtomo_Rec(Lista_Gen *L, ITEM *x)
- // Busca ocorrência do átomo x na lista generalizada L; retorna seu endereço. Devolve null se não achou
- int Prof(Lista_Gen *S)
- // Calcula a profundidade da lista generalizada S
- *Lista_Gen Copia(Lista_Gen *L)
- // cria uma copia da lista generalizada L

Operações úteis

- int Igual(Lista_Gen *S, Lista_Gen *T)
- // Verifica a igualdade de duas listas generalizadas S e T
- void imprimir(tNO *L)
- // Imprime a lista L
- void apagar(tNO *L)
- //apagar uma lista, liberando a sua memória alocada

Operações

```
void cria(Lista_Gen *L){
  L->noptr = NULL;
}
```

```
void concatena(Lista_Gen *L1,Lista_Gen *L2){
    if (L1->noptr==NULL)
      L1->noptr = L2->noptr;
    else {
      tNO *p = L1->noptr;
      while (p->prox != NULL)
         p = p->prox;
      p->prox = L2->noptr;
    }
}
```

Operações

```
void imprime(tNO *L){
  if (L != NULL)
      if (L->tipo == 0) {
         printf("%c ", L->info.atomo);
         imprime(L->prox);
      }else {
        printf(" (");
        imprime(L->info.sublista);
        printf(") ");
        imprime(L->prox);
 }
}
```

Operações

```
void apagar(tNO *L){
   tNO *t;
   if (L != NULL) {
      if (L->tipo == 1) {
         apagar(L->info.sublista);
         t = L;
         L = L->prox;
         free(t);
         apagar(L);
       }else {
         t = L;
         L = L->prox;
         free(t);
         apagar(L);}
   }
}
```

Operações de Inserção de Elementos

- // Insere o primeiro atomo numa Lista generalizada.
- void Insere_Prim_Atomo(Lista_Gen *L, elem *dado);
- // Insere uma lista numa Lista generalizada.
- void Insere_Lista(Lista_Gen *L, Lista_Gen L1);
- // Insere um atomo no início de uma Lista generalizada.
- void Insere_Inicio_Atomo(Lista_Gen *L, elem *dado);
- // Insere uma lista no início de uma Lista generalizada.
- void Insere_Inicio_Lista(Lista_Gen *L, Lista_Gen L1);

Lista Generalizada

- Exercícios
- Implementar uma função recursiva para buscar um átomo x numa lista generalizada
  - (1) considere apenas a lista principal;
  - (2) considere que x pode estar em qualquer sublista.
- Implementar uma sub-rotina para verificar se duas listas generalizadas são iguais
  - Tente fazer a sub-rotina recursiva

Algoritmos

- Uma função booleana recursiva para buscar um átomo x numa lista generalizada, L. Retorna também o endereço, se estiver lá.
  - (1) considere apenas a lista principal;

Algoritmos

```
int esta_na_lista_principal(tNO *p, char x) {
   int achou=0;
   while ((p!=NULL) && (!achou)) {
      if ((p->tipo==0) && (p->info.atomo==x))
        achou=1;
      else p=p->prox;
   }
   return(achou);
}
```

Algoritmos

- Uma função booleana recursiva para buscar um átomo x numa lista generalizada, L. Retorna também o endereço, se estiver lá.
  - (2) considere que x pode estar em qualquer sublista

Algoritmos

```
int esta_em_qualquer_parte_da_lista(tNO *p, char x) {
  if (p==NULL)
     return 0;
  else if (p->tipo==0) {
    if (p->info.atomo==x)
       return 1;
    else
       return(esta_em_qualquer_parte_da_lista(p->prox,x));}
 else if (p->tipo==1) {
    if (esta_em_qualquer_parte_da_lista(p->info.sublista,x))
      return 1;
    else
      return(esta_em_qualquer_parte_da_lista(p->prox,x));
   }
}
```

Algoritmos

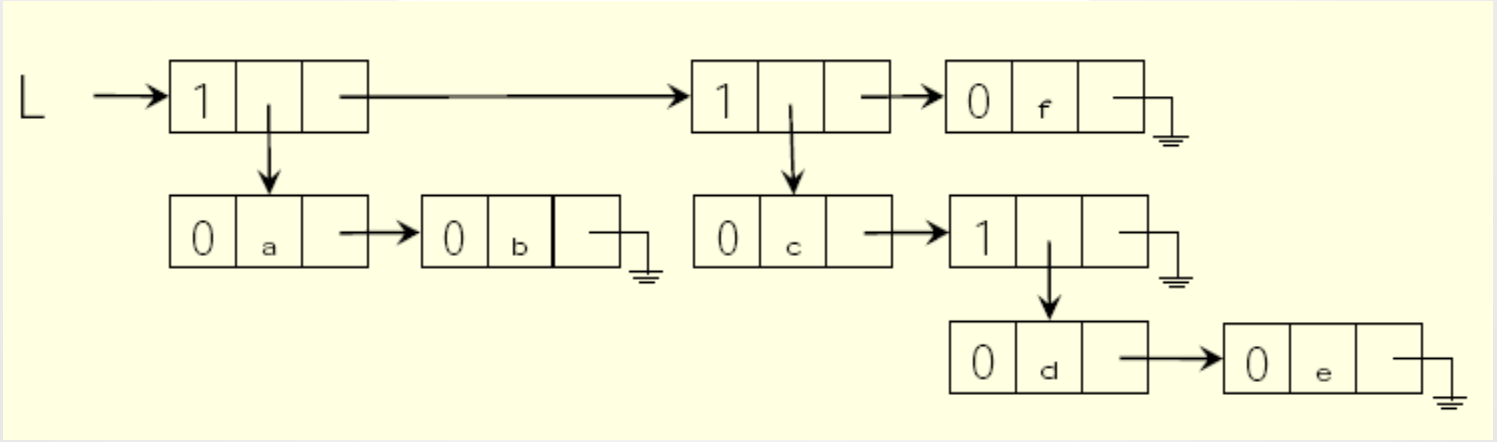
- Verificar se duas listas generalizadas, L1 e L2, são iguais
  - Tente fazer função booleana recursiva
  - O conteúdo em si não importa
  - 4 casos:
    - Caso 1: 2 listas vazias (true)
    - Caso 2: uma das listas é vazia e a outra não (false)
    - Caso 3: 2 listas não vazias
      - Átomos – se Iguais - Testa_cauda (true)
      - 2 listas –se Iguais_Testa cabeça_lista se Iguais_Testa cauda (true)
    - Caso 4: se diferem na cabeça_lista(false)

Algoritmos

```
int iguais(no *L1, no *L2) {
  if ((L1==NULL) && (L2==NULL))
     return 1;
  else if ((L1==NULL) || (L2==NULL))
    return 0;
  else if
((L1->tipo==L2->tipo==0)&&(L1->info.atomo==L2->info.atomo))
    return(iguais(L1->prox,L2->prox));
  else if
((L1->tipo==L2->tipo==1)&&(iguais(L1->info.sublista,L2->info.
sublista)))
   return(iguais(L1->prox,L2->prox));
  else
      return 0;
}
```

Listas e recursão

- Exercício extra
  - Implementar uma sub-rotina que determina a profundidade máxima de uma lista generalizada
  - Tente usar recursividade
  - Por exemplo, para o caso abaixo, a sub-rotina deveria retornar profundidade 3



Profundidade máxima de uma lista generalizada S

- Ex. S = (a, (b)) => Prof(S) = 2

  A = (a, b, c) => Prof(A) = 1

  B = () => Prof(B) = 0;

Algoritmos

```
int profundidade(tNO *p) {
    int prof, aux;
    if (p==NULL)
      prof=0;
    else if (p->tipo==0) {
      prof=1;
      aux=profundidade(p->prox);
      if (aux>prof)
        prof=aux; }
    else if (p->tipo==1) {
      prof=1+profundidade(p->info.sublista);
      aux=profundidade(p->prox);
      if (aux>prof)
         prof=aux;
      }
    return(prof);
}
```

Algoritmos

- Exercício
  - Implementar uma sub-rotina para verificar se duas listas generalizadas são estruturalmente iguais
    - O conteúdo em si não importa

Algoritmos

```
int iguais_estruturalmente(tNO *L1, tNO *L2) {
  if ((L1==NULL) && (L2==NULL))
    return 1;
  else if ((L1==NULL) || (L2==NULL))
    return 0;
  else if (L1->tipo==L2->tipo==0)
    return(iguais_estruturalmente(L1->prox,L2->prox));
  else if ((L1->tipo==L2->tipo==1) &&
(iguais_estruturalmente(L1->
info.sublista,L2->info.sublista)))
   return(iguais_estruturalmente(L1->prox,L2->prox));
  else
    return 0;
}
```

Lista generalizada e polinômios

- Considere os polinômios:

Considere os polinômios:

$P1 = 4x^2y^3z + 3xy + 5$

$P2 = x^{10}y^3z^2 + 2x8y^2z^2 + x4y4z + 6x^3y4z + 2yz$

$P3 = 3x^2y$

(a) nº de termos: variável

- P1=3, P2=5, P3=1

(b) nº de variáveis: variável

- P1=P2=3, P3=2

(c) nem todo termo é expresso com todas as variáveis

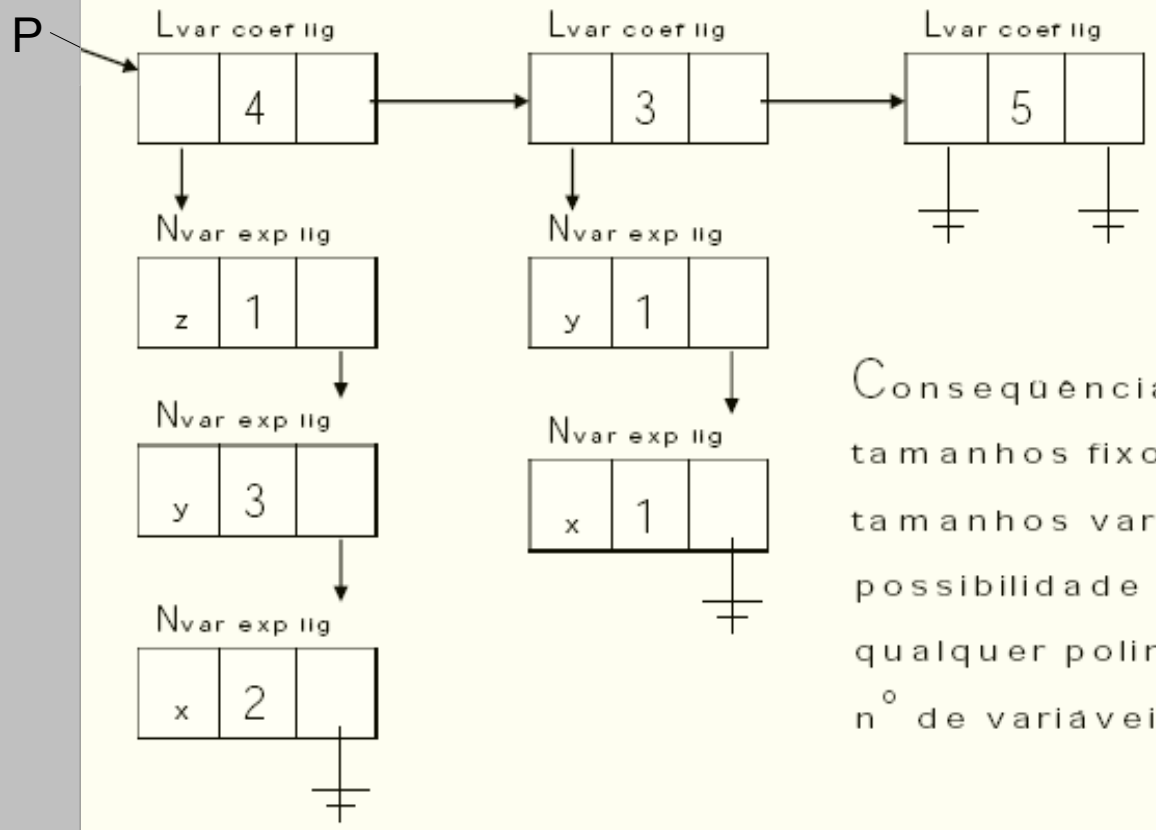
Lista generalizada e polinômios

- Objetivos
  - representar de forma a otimizar o uso de memória
  - representação única para todo polinômio

- Solução: lista generalizada

Uso de lista generalizada para representação de polinômios –Opção 1 –2 tipos de registros

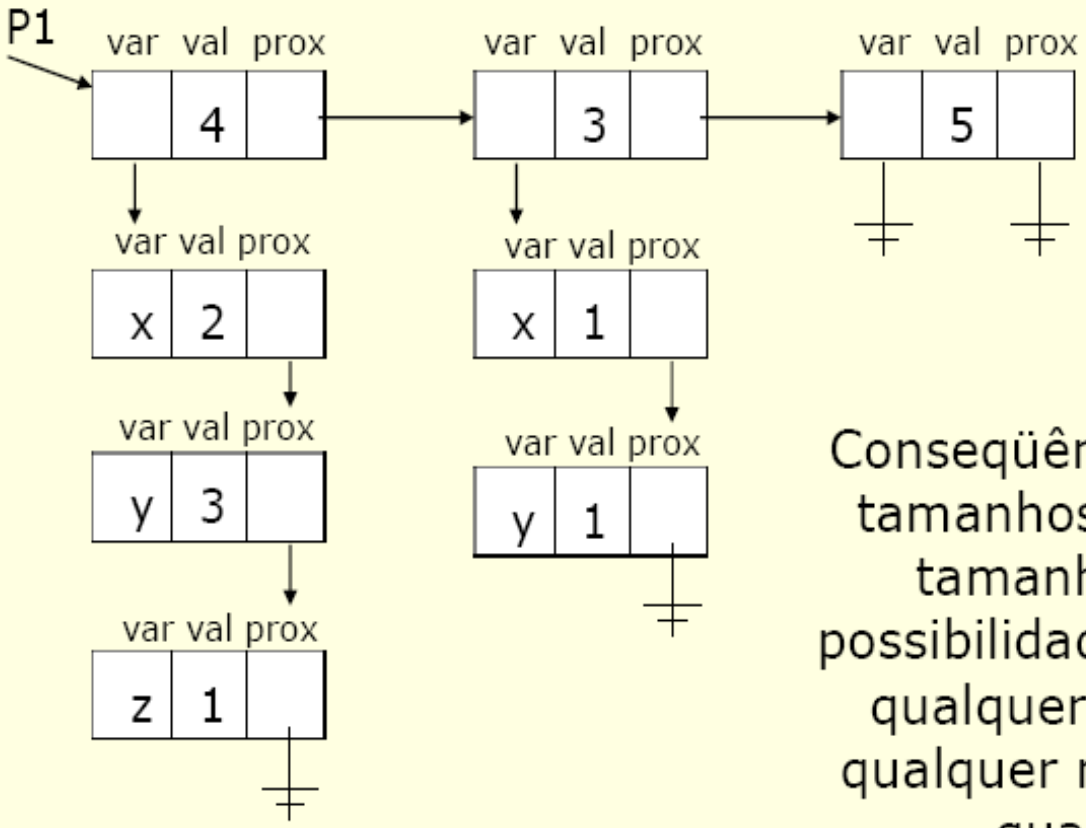
Ex: (1) $P(x,y,z) = 4x^2y^3z+3xy + 5$



Consequência: registros de tamanhos fixos; listas de tamanhos variáveis; possibilidade de representar qualquer polinômio em qualquer nº de variáveis, qualquer grau

Uso de lista generalizada para representação de polinômios –Opção 2 –1 tipos de registros

Ex: P1 = $4x^2y^3z + 3xy + 5$



Conseqüência: registros de tamanhos fixos; listas de tamanhos variáveis; possibilidade de representar qualquer polinômio com qualquer n° de variáveis e qualquer grau

Exercícios

- Faça a declaração dos tipos das 2 opções anteriores
- Implementar uma função que:
  - (a) receba um polinômio representado via lista generalizada e os valores das variáveis
  - (b) percorra a lista generalizada e compute o resultado do polinômio
  - (c) retorne o resultado para quem chamou a sub-rotina