Estruturas de Dados II

Professor: Francisco Assis da Silva

Listas Generalizadas

PEREIRA, Silvio do Lago, Estruturas de Dados Fundamentais, Ed. Érica. págs. 157 a 171

Fundamentos

Uma **lista generalizada** L: **[e_1, e_2, e_3, ...,e_n]**, **n** ≥ **0** , é uma coleção onde cada elemento **e_k**, **1** ≤ **k** ≤ **n** é uma partícula indivisível (**átomo**) ou então é uma outra lista generalizada (**sublista**). O comprimento de uma lista generalizada L é definido como sendo o valor **n**. Se **n=0**, dizemos que a lista está vazia (**nula**), caso contrário, **e_1** é a **cabeça** da lista e **[e_2, e_3, ...,e_n]** é a sua **cauda.**

Exemplos de Listas Generalizadas

[] é uma lista vazia cujo comprimento é 0;
[a, b] tem comprimento 2 e contém os átomos **a** e **b**;
[a, [b, c]] tem comprimento 2 e contém o átomo **a** e a lista **[b, c]**;
[a, [b , [c]], d] tem comprimento 3 e contém os átomos **a**, **d**, e a lista **[b, [c]]**;
[[]] é uma lista de comprimento 1, contendo a lista nula **[]**.

As principais operações sobre listas generalizadas são: **Head()** e **Tail()**. Se **L** é uma lista não nula, então **Head(L)** é o valor do primeiro elemento de **L**, que pode tanto ser um átomo quanto uma lista; **Tail(L)** é a lista, que pode ser nula. Se **L** é uma lista vazia (ou um átomo), **Head(L)** e **Tail(L)** não têm valor definido.

Exemplo de funcionamento das operações **Head()** e **Tail()**
L:[[Elis, [olhos, azuis]], [Yara, [idade, 23]]]

Head(L) ≡ [Elis, [olhos, azuis]]
Head(Head(L)) ≡ Elis
Tail(L) ≡ [[Yara, [idade, 23]]]
Head(Tail(L)) ≡ [Yara, [idade, 23]]
Tail(Head(Tail(L))) ≡ [[idade, 23]]
Head(Tail(Head(Tail(L)))) ≡ [idade, 23]
Tail(Head(Tail(Head(Tail(L))))) ≡ [23]
Head(Tail(Head(Tail(Head(Tail(L)))))) ≡ 23

Tanto as operações Head() e Tail() podem resultar numa lista generalizada, entretanto apenas Head() pode resultar num átomo.

Para construir uma lista generalizada:

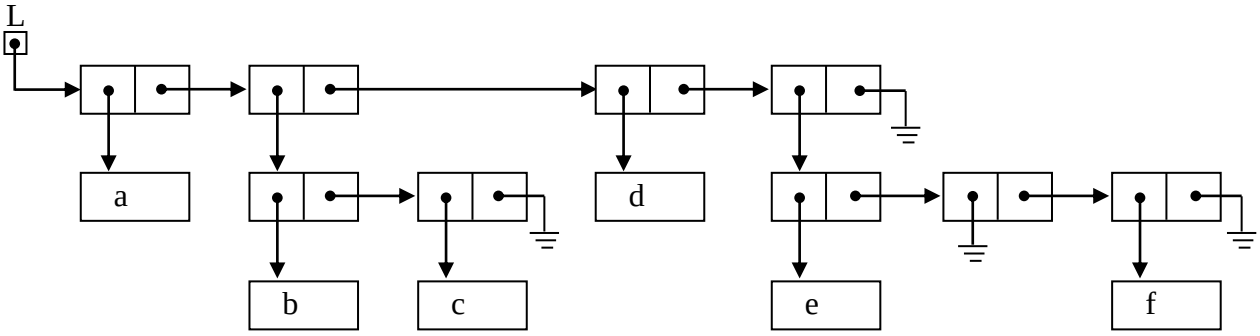
Operação construtora básica Cons().
A operação Cons(H, T) constrói uma lista cuja cabeça é H e cuja cauda é T.
Exemplos:

Cons(a, []) ≡ [a]
Cons([], []) ≡ [[]]
Cons(b, Cons(a, [])) ≡ [b, a]
Cons(Cons(a, []), Cons(b, [])) ≡ [[a], b]
Cons(Cons(a, Cons(b, [])), Cons(c, [])) ≡ [[a, b], c]

Representação de Listas Generalizadas

Dois tipos básicos de nodos:

- **Terminais:** nodos que armazenam os átomos contidos na lista generalizada;
- **Não-terminais:** nodos que descrevem a forma da lista, isto é, a estrutura de aninhamento e relacionamento dos átomos nela contidos.



Representação interna da lista generalizada L: [a, [b, c], d, [e, [], f]]

Nodos não-terminais são compostos por dois campos, um que armazena um ponteiro para baixo (**cabeça**) e outro que armazena um ponteiro para frente (**cauda**).

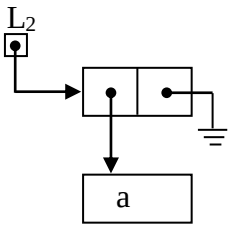
A estrutura da lista generalizada possui dois níveis básicos:

- **Superior:** uma lista encadeada, acessível através do ponteiro inicial L, que possui um nodo não-terminal para cada um dos elementos da lista;
- **Inferior:** acessível através dos "ponteiros para baixo" armazenados no nível superior, e pode conter uma mistura de nodos terminais e não-terminais. Se um nodo do nível superior representa um átomo, então seu ponteiro para baixo acessa um nodo terminal, caso contrário, seu ponteiro acessa uma lista generalizada (contendo os mesmos dois níveis básicos).

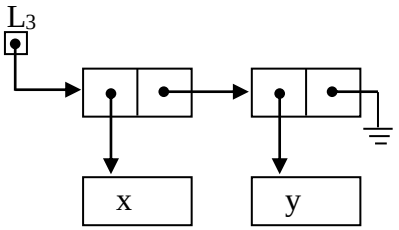
Representação interna para algumas listas generalizadas:

L_1: []

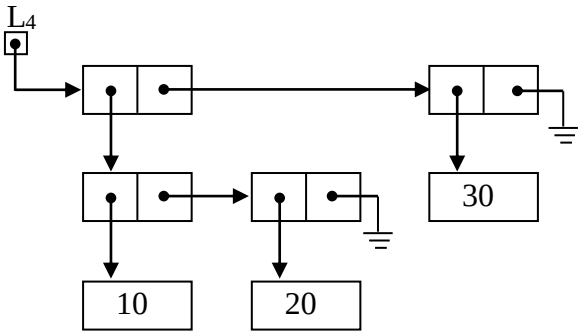
L_2: [a]



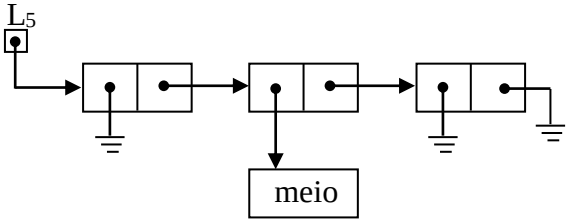
L_3: [x, y]



L_4: [[10, 20], 30]



L_5: [[], meio, []]



Exercícios

1:-) Esboce a representação interna para cada uma das listas generalizadas:

a) [a, [b, [c, [d]]]]
b) [[a, [b, c]], d]
c) [[a, b, c], [d, e], [], f]
d) [[[a]]]
e) [[[[]]]]

2:-) Considere a lista L: [[a, b], c, [d, [e, f], g]]. Usando as operações Head() e Tail(), dê exemplos capazes de acessar os seguintes elementos de L:

a) [a, b]
b) c
c) g
d) f

3:-) Considere as seguintes listas generalizadas

L: [[[[a], b], c, [d, e, [f, g]]]]
M: [1, [2, 3], [[[4], 5], 6], 7]

Usando as operações Head(), Tail() e Cons(), dê expressões para construir as listas a seguir a partir das listas L e M. Dica: use variáveis auxiliares para construir listas intermediárias.

a) [a, 1]
b) [[a, 1], [b, 2]]
c) [[c, d], [3, 4]]
d) [a, b, c, d, e, f, g]
e) [1, 2, 3, 4, 5, 6, 7]

Implementação de Listas Generalizadas

O nodo cabeça pode apontar tanto nodos terminais quanto não-terminais. Para que isso possa acontecer, usamos um recurso da linguagem C chamado **union** (em Pascal registro variante). Um registro variante ora pode ter um formato (um conjunto de campos) e ora pode ter outro, dependendo do valor de um campo especial chamado etiqueta.

Exemplo de **union**

```
#include <stdio.h>

struct bits
{                 - significativo
  unsigned char b7:1;
  unsigned char b6:1;
  unsigned char b5:1;
  unsigned char b4:1;
  unsigned char b3:1;
  unsigned char b2:1;
  unsigned char b1:1;
  unsigned char b0:1;
};                + significativo

union byte
{
  struct bits bi;
  unsigned char num;
};

main(void)
{
  union byte uval;
  uval.num = 12;
  printf("%d",uval.bi.b0);
  printf("%d",uval.bi.b1);
  printf("%d",uval.bi.b2);
  printf("%d",uval.bi.b3);
  printf("%d",uval.bi.b4);
  printf("%d",uval.bi.b5);
  printf("%d",uval.bi.b6);
  printf("%d",uval.bi.b7);
  getch();
}
```

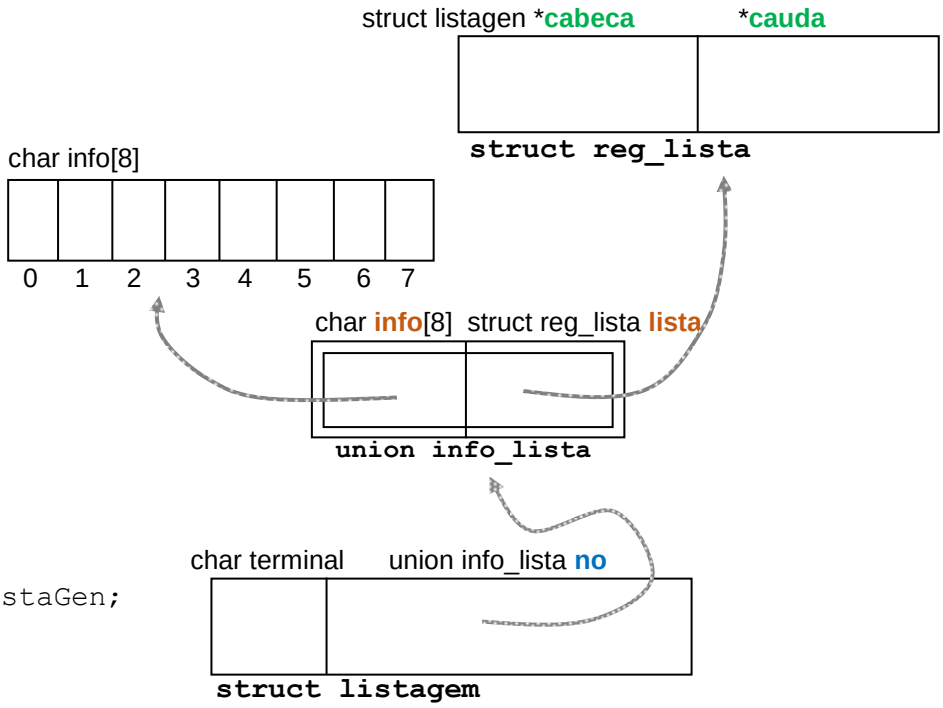
Declaração da Estrutura de Dados

Em **C**:

```
struct reg_lista
{
  struct listagen *cabeca;
  struct listagen *cauda;
};

union info_lista
{
  char info[8];
  struct reg_lista lista;
};

struct listagen
{
  char terminal;
  union info_lista no;
};
typedef struct listagen ListaGen;
```



Em **Pascal**:

```
type TpInfo = string[8];
     ListaGen = ^nodo;
     Nodo = record
              case terminal:boolean of
                false: (cabeca, cauda: ListaGen);
                true: (info: TpInfo);
            end;
```

O campo etiqueta, chamado **terminal**, é uma variável lógica que pode assumir o valor verdadeiro (1) ou falso (0). Se tiver falso, significa que o registro é do tipo não-terminal, possuindo os campos **cabeca** e **cauda**. Caso contrário, o registro é do tipo terminal e possui apenas o campo **info**. Em termos de alocação de memória, todos os registros utilizam a mesma quantidade de bytes, o compilador sempre se baseia no maior campo (maior quantidade de bytes).

Representação gráfica dos nodos com registros variantes

Algoritmos para construção de Listas Generalizadas

→ Para criar um **átomo**, precisamos alocar um **nodo terminal** e armazenar o elemento no campo **info**.

```
ListaGen *CriaT(char *info)
{
  ListaGen *L = (ListaGen*)malloc(sizeof(ListaGen));
  L->terminal = 1;
  strcpy(L->no.info, info);
  return L;
}
```

→ Para verificar se um ponteiro aponta para uma **lista** ou apenas um **átomo**.

```
char Nula(ListaGen *L)
{
  return L==NULL;
}

char Atomo(ListaGen *L)
{
  return !Nula(L) && L->terminal;
}
```

→ Para construir listas generalizadas.
O Segundo argumento deve ser obrigatoriamente uma **lista**. Caso a função receba um **átomo**, uma mensagem de erro é dada e a função retorna uma **lista nula**.

```
ListaGen *Cons(ListaGen *H, ListaGen *T)
{
  if (Atomo(T))
  {
    printf("Cons: Segundo arqumento nao pode ser Atomo!");
    return NULL;
  }
  else
  {
    ListaGen *L = (ListaGen*)malloc(sizeof(ListaGen));
    L->terminal = 0;
    L->no.lista.cabeca = H;
    L->no.lista.cauda = T;
    return L;
  }
}
```

Exemplos de uso das funções CriaT() e Cons():

Cons(CriaT(“único”), NULL) ≡ [único]
Cons(CriaT(“um”), Cons(CriaT(“dois”), NULL)) ≡ [um, dois]
Cons(Cons(CriaT(“x”), NULL), Cons(CriaT(“y”), NULL)) ≡ [[x], y]

→ Função que retorna a **cabeça** de uma **lista**:

```
ListaGen *Head(ListaGen *L)
{
  if (Nula(L) || Atomo(L))
  {
    printf("Head: argumento deve ser lista não vazia!");
    return NULL;
  }
  else
    return L->no.lista.cabeca;
}
```

→ Função que retorna a **cauda** de uma **lista**:

```
ListaGen *Tail(ListaGen *L)
{
  if (Nula(L) || Atomo(L))
  {
    printf("Tail: argumento deve ser lista não vazia!");
    return NULL;
  }
  else
    return L->no.lista.cauda;
}
```

Exercícios:

4:-) Usando as operações Cons() e Criat(), escreva expressões para construir as seguintes listas:

a) [a, b, c]
b) [a, [b, [c]]]
c) [[[a], b], c]
d) [[a, b, [c]], d]
e) [[[[]]]]

→ Algoritmo para exibir uma lista generalizada **L**:
Duas possibilidades:
- L aponta para um átomo:
- L aponta para uma lista generalizada.

→ Algoritmo para destruir uma lista generalizada (liberar todos os nodos de uma lista)
Três casos a considerar:
- L tem valor nulo, representando uma lista vazia;
- L aponta um nodo terminal, representando um átomo;
- L aponta um nodo não terminal, representando uma lista generalizada.

→ Algoritmo para duplicar uma lista generalizada.

→ Algoritmo capaz de comparar duas listas generalizadas.