Linguagens de Programação
Conceitos e Técnicas

Valores e Tipos de Dados

Conceituação

- Um valor é uma entidade que existe durante uma computação

- Valor (= Dado)

  3 2.5 'a' "Paulo" 0x1F 026

Conceituação

- Um tipo de dado é um conjunto cujos valores exibem comportamento uniforme nas operações associadas ao tipo
- Possuem cardinalidade
- Tipo (de dado)

  { true, 25, 'b', "azul" } não corresponde a um tipo

  { true, false } corresponde a um tipo

Importância de um sistema de tipos

Durante a escrita e a depuração, adquiri um grande respeito pela expressividade do sistema de tipos de Simula e pela capacidade do seu compilador de capturar erros de tipos. Observei que erros de tipos quase invariavelmente refletiam um erro tolo de programação ou uma falha conceitual no projeto. [...] Em contraste, tinha descoberto que o sistema de tipos de Pascal era pior do que inútil – uma camisa-de-força que causava mais problemas do que soluções, forçando-me a entortar meus projetos [...]

Stroustrup, criador do C++ [T&N]

Tipos Primitivos

- Não podem ser decompostos em valores mais simples
- Costumam ser definidos na implementação da LP
  - Sofrem influência direta do hardware (ex.: int em C, diferente em plataformas diferentes)
  - Podem indicar o propósito da linguagem (ex.: COMPLEX em Fortran)

Tipo Inteiro

- Corresponde a um intervalo do conjunto dos números inteiros
- Vários tipos inteiros numa mesma LP
  - Normalmente, intervalos são definidos na implementação do compilador
- Em JAVA, o intervalo de cada tipo inteiro é estabelecido na definição da própria LP

Tipos Inteiros em JAVA

Tipo	Tamanho (bits)	Intervalo	
		Início	Fim
byte	8	-128	127
short	16	-32768	32767
int	32	-2.147.483.648	2.147.483.647
long	64	-9223372036854775808	9223372036854775807

Tipo Caractere

- Armazenados como códigos numéricos
  - Tabelas EBCDIC, ASCII e UNICODE
- PASCAL e MODULA 2 oferecem o tipo *char*
- Em C, o tipo primitivo *char* é classificado como um tipo inteiro

```
char d;
char *p, *q;
d = 'a' + 3;
...
while (*p) *q++ = *p++;
```

Tipo Booleano

- Tipo mais simples
  - Possui apenas dois valores
- C não possui o tipo de dado booleano, mas qualquer expressão numérica pode ser usada como condicional

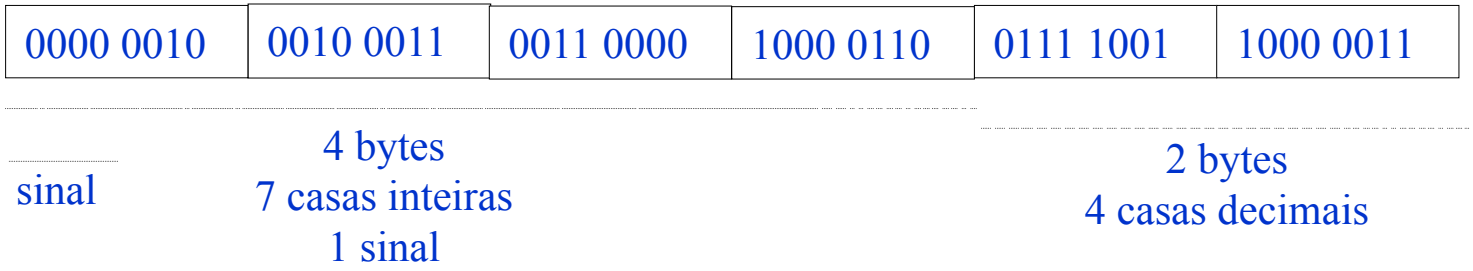
  Valores ≠ zero ⇒ verdadeiro

  Valores = zero ⇒ falso
  - Abordagem de C pode provocar erros

    ```
    if (c += 1) x = 10;
    ```
- JAVA inclui o tipo de dado *boolean*

Tipo Decimal

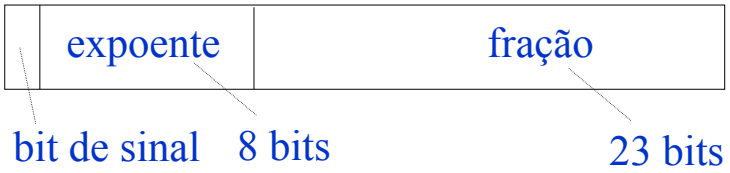
- Armazena um número fixo de dígitos decimais
  - COBOL possui



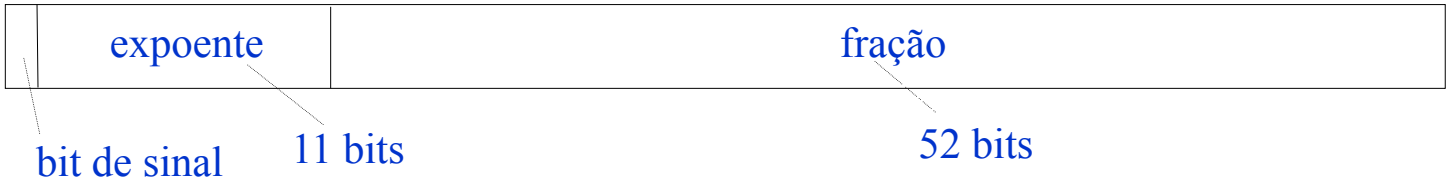
  - Vantagem: precisão;
  - Desvantagem: reduzido intervalo de valores, desperdício de memória.

Tipo Ponto Flutuante

- O tipo primitivo ponto flutuante modela os números reais
- LPs normalmente incluem dois tipos de ponto flutuante: *float* e *double*



Padrão IEEE 754 - Precisão Simples



Padrão IEEE 754 - Precisão Dupla

Tipo Enumerado

- PASCAL, ADA, C e C++ permitem que o programador defina novos tipos primitivos através da enumeração de identificadores dos valores do novo tipo

```
enum mes_letivo { mar, abr, mai, jun, ago, set, out, nov };

enum mes_letivo m1, m2;
```

- Possuem correspondência direta com intervalos de tipos inteiros e podem ser usados para indexar vetores e para contadores de repetições
- Aumentam a legibilidade e confiabilidade do código
- Java não suportava tipos enumerados até a versão 5

Tipo Intervalo de Inteiros

- Em PASCAL e ADA, também é possível definir tipos intervalo de inteiros

```
type meses = 1 .. 12;
```

- Tipos intervalos herdam as operações dos inteiros

Tipos Compostos

- Tipos compostos são aqueles que podem ser criados a partir de tipos mais simples

  registros, vetores, listas, arquivos

- Entendidos em termos dos conceitos

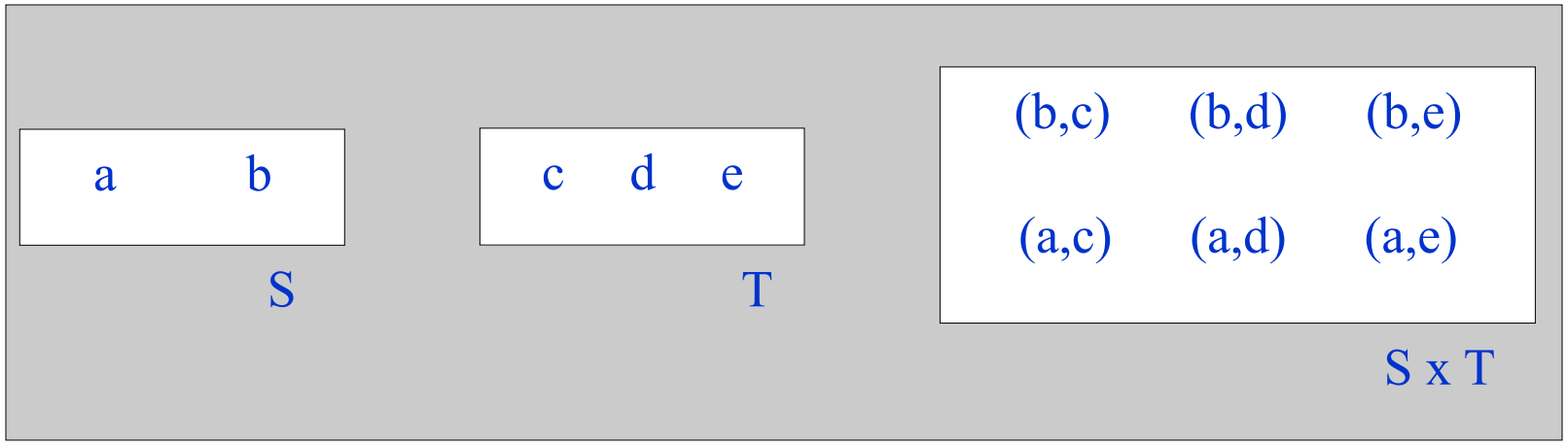
  produto cartesiano, uniões, mapeamentos, conjuntos potência e tipos recursivos

- Cardinalidade

  número de valores distintos que fazem parte do tipo

Produto Cartesiano

- Combinação de valores de tipos diferentes em tuplas



Produto Cartesiano

- São produtos cartesiano os registros de PASCAL, MODULA 2, ADA e COBOL e as estruturas de C

```
struct nome {
    char primeiro [20];
    char meio [10];
    char sobrenome [20];
}
struct empregado {
    struct nome nfunc;
    float salario;
} emp;
```

Produto Cartesiano

- Uso de seletores

  emp.nfunc.meio

- Inicialização em C

  struct data { int d, m, a; };

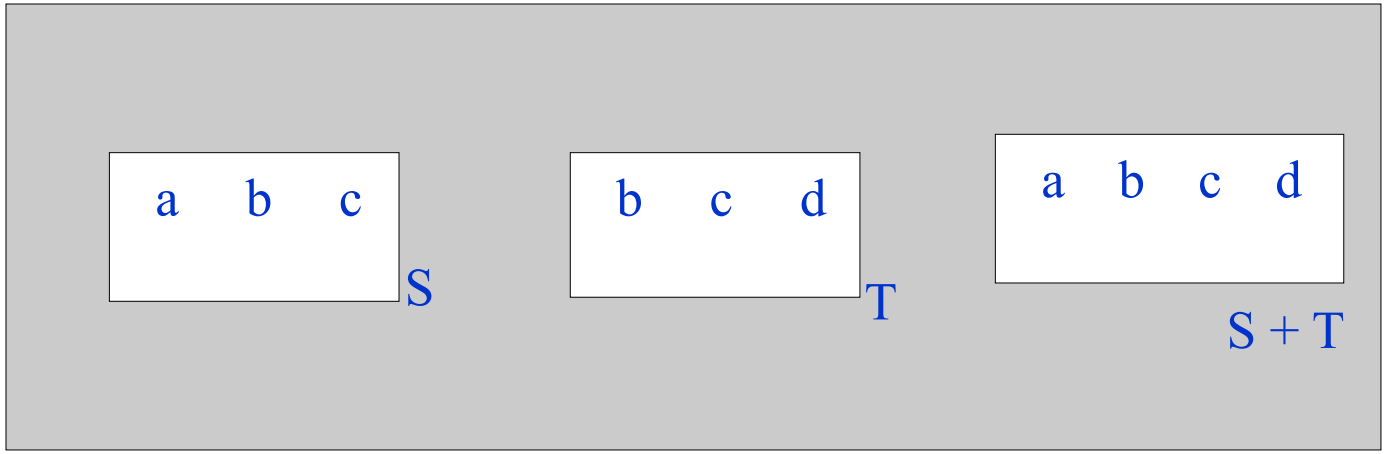
  struct data d = { 7, 9, 1999 };

- Em LPs orientadas a objetos, produtos cartesianos são definidos a partir do conceito de classe
  - JAVA só tem class
- Cardinalidade

  #(S1 x S2 x ... x Sn) = #S1x #S2x ...x #Sn

Uniões

- Consiste na união de valores de tipos distintos para formar um novo tipo de dados



Uniões

- Uniões Livres
  - Pode haver interseção entre o conjunto de valores dos tipos que formam a união
  - EQUIVALENCE de Fortran e union de C
  - Há possibilidade de violação no sistema de tipos

```
union medida {
  int centimetros;
  float metros;
};
```

```
union medida medicao;
float altura;
medicao.centimetros=180;
altura = medicao.metros;
printf("\n altura : %f metros\n", f);
```

Uniões

- Uniões Disjuntas
  - não há interseção entre o conjunto de valores dos tipos que formam a união
  - registros variantes de PASCAL, MODULA 2 e ADA e a union de ALGOL 68

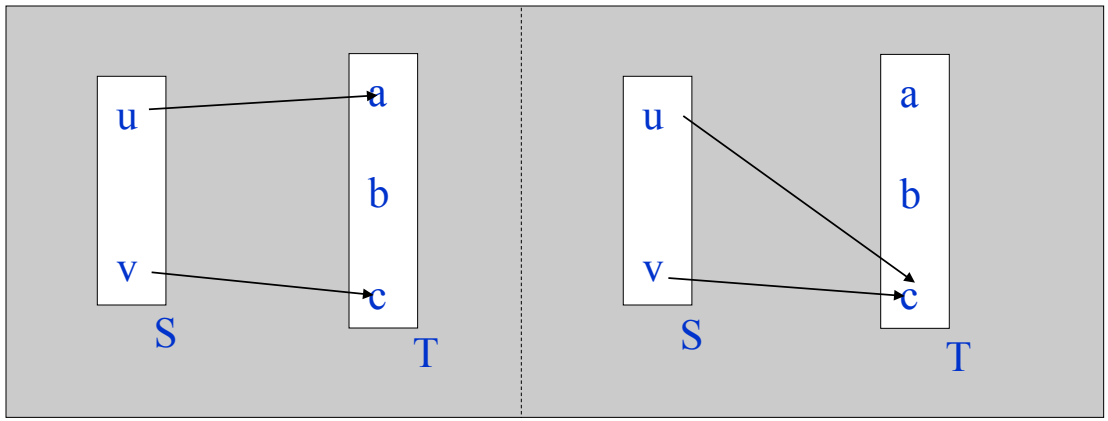
```
TYPE Representacao = (decimal, fracionaria);
        Numero = RECORD CASE Tag: Representacao OF
            decimal: (val: REAL);
            fracionaria: (numerador, denominador: INTEGER);
        END;
```

  - Cardinalidade

$$\#(S_1 + S_2 + \ldots + S_n) = \#S_1 + \#S_2 \ldots + \#S_n$$

Mapeamentos

- Tipos de dados cujo conjunto de valores corresponde a todos os possíveis mapeamentos de um tipo de dados S em outro T



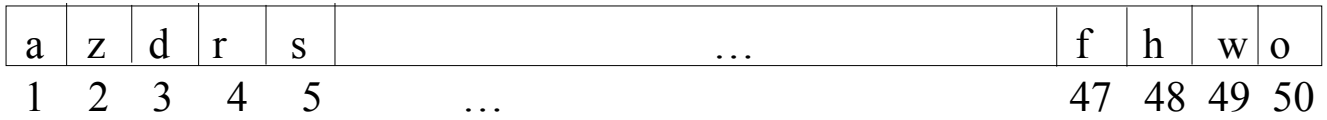
- Cardinalidade: $\#(S \rightarrow T) = (\#T)^{\#S}$

Mapeamentos Finitos

- O conjunto domínio é finito
- Vetores e matrizes

  array S OF T; (S → T)

  A: array [1.50] of char; A: ([1,50] → char)



- O conjunto índice deve ser finito e discreto
- Verificação de Índices em C e JAVA

```
int v[7];
V[13] = 198;
```

Categorias de Vetores

Categoria de Vetor	Tamanho	Tempo de Definição	Alocação	Local de Alocação	Exemplos
Estáticos	Fixo	Compilação	Estática	Base	FORTRAN 77
Semi-Estáticos	Fixo	Compilação	Dinâmica	Pilha	PASCAL, C, MODULA 2
Semi-Dinâmicos	Fixo	Execução	Dinâmica	Pilha	ALGOL 68, ADA, C
Dinâmicos	Variável	Execução	Dinâmica	Monte	APL, PERL

Categorias de Vetores

- Estáticos (C)

```
void f () {
    static int x[10];
}
```

- Semi-Estáticos (C)

```
void f () {
    int x[10];
}
```

- Semidinâmicos (Padrão ISO - 1999)

```
void f (int n) {
    int x[n];
}
```

- Dinâmicos (APL)

```
A ← (2  3  4)
A ← (2  3  4  15 )
```

Vetores Dinâmicos

- Podem ser implementados em C, C++ e JAVA através do monte
- Necessário alocar nova memória e copiar conteúdo quando vetor aumenta de tamanho
- É encargo do programador controlar alocação e cópia. Em C e C++, o programador deve controlar desalocação também. Isso torna a programação mais complexa e suscetível a erros

Vetores Multidimensionais

- Elementos são acessados através da aplicação de fórmulas

  posição mat [i] [j] =

  endereço de mat [0][0] + i × tamanho da linha + j × tamanho do elemento =

  endereço de mat [0][0] + (i × número de colunas + j) × tamanho do elemento

Vetores Multidimensionais

- Em JAVA vetores multidimensionais são vetores unidimensionais cujos elementos são outros vetores

```
int [] [] a = new int [5] [];
for (int i = 0; i < a.length; i++) {
    a [i] = new int [i + 1];
}
```

- O mesmo efeito pode ser obtido em C com o uso de ponteiros para ponteiros

Vetores Multidimensionais

- Cardinalidade

  int mat [5][4];

  Conjunto de valores

  $\{0, \ldots, 4\} \times \{0, \ldots, 3\} \rightarrow int$

  $(\#int)^{\#(\{0, \ldots, 4\} \times \{0, .., 3\})} =$

  $(\#int)^{(\#\{0, \ldots, 4\} \times \#\{0, .., 3\})} =$

  $(\#int)^{5 \times 4} =$

  $(\#int)^{20}$

Mapeamentos Através de Funções

- Uma função implementa um mapeamento S→T através de um algoritmo
- O conjunto S não necessita ser finito
- O conjunto de valores do tipo mapeamento S→T são todas as funções que mapeiam o conjunto S no conjunto T
- Valores do mapeamento [int → boolean] em JAVA

```
boolean positivo (int n) {
        return n > 0;
}
```

Outros valores do mapeamento: palindromo, impar, par, primo

Mapeamentos Através de Funções

- C utiliza o conceito de ponteiros para manipular endereços de funções como valores

```
int impar (int n){ return n%2; }
int negativo (int n) { return n < 0; }
int multiplo7 (int n) { return !(n%7); }
int conta (int x[], int n, int (*p) (int) ) {
    int j, s = 0;
    for (j = 0; j < n; j++)
        if ( (*p) (x[j]) ) s++;
    return s;
}
```

Mapeamentos Através de Funções

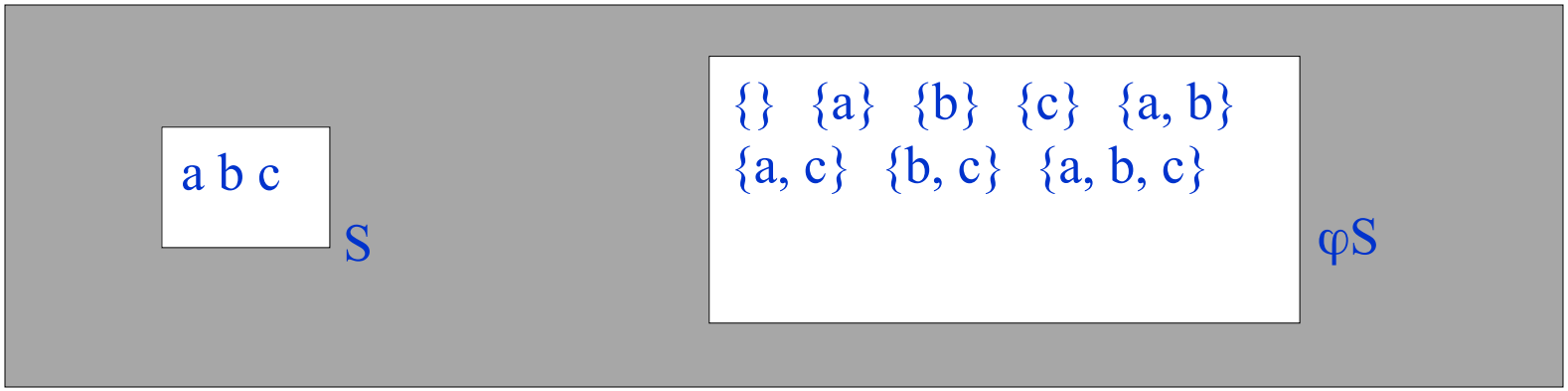
```
main() {
    int vet [10];
    printf ("%d\n", conta (vet, 10, impar));
    printf ("%d\n", conta (vet, 10, negativo));
    printf ("%d\n", conta (vet, 10, multiplo7));
}
```

- JAVA não trata funções (métodos) como valores
- Pode-se empregar algoritmos diferentes para implementar um mesmo mapeamento
- Algumas vezes, vetores e funções podem ser usados para implementar o mesmo mapeamento finito

Conjuntos Potência

- Tipos de dados cujo conjunto de valores corresponde a todos os possíveis subconjuntos que podem ser definidos a partir de um tipo base S

  $\varphi S = \{s \mid s \subseteq S\}$



Conjuntos Potência

- Cardinalidade
  - $\#\varphi S = 2^{\#S}$
- Operações básicas
  - Pertinência
  - Contém
  - Está contido
  - União
  - Diferença
  - Diferença simétrica
  - Interseção

Conjuntos Potência

- Poucas LPs oferecem. Muitas vezes de forma restrita
- PASCAL (somente discretos, primitivos e pequenos):

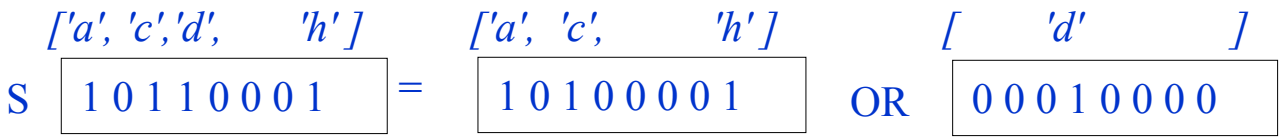
```
TYPE
      Carros = (corsa, palio, gol);
      ConjuntoCarros = SET OF Carros;
VAR
    Carro: Carros;
    CarrosPequenos: ConjuntoCarros;
BEGIN
    Carro:= corsa;
    CarrosPequenos := [palio, gol];                    /*atribuicao*/
    CarrosPequenos:= CarrosPequenos + [corsa];  /*uniao*/
```

Conjuntos Potência

```
CarrosPequenos:= CarrosPequenos * [gol];      /*intersecao*/
if Carro in CarrosPequenos THEN               /*pertinencia*/
if  CarrosPequenos >= [gol, corsa] THEN       /*contem*/
```

- Restrições de PASCAL visam permitir implementação eficiente

```
VAR S: SET OF [ 'a .. 'h' ];
 BEGIN
    S := ['a', 'c', 'h'] + ['d'];
 END;
```



Tipos Recursivos

- Tipos recursivos são tipos de dados cujos valores são compostos por valores do mesmo tipo

  R ::= < parte inicial > R < parte final >

  Tipo Lista ::= Tipo Lista Vazia | (Tipo Elemento x Tipo Lista)
- A cardinalidade de um tipo recursivo é infinita
- Isto é verdade mesmo quando o tipo do elemento da lista é finito
- O conjunto de valores do tipo listas é infinitamente grande (não podendo ser enumerado) embora toda lista individual seja finita

Tipos Recursivos

- Tipos recursivos podem ser definidos a partir de ponteiros ou diretamente

Em C

```
struct no {
   int  elem;
   struct no* prox;
};
```

Em C++

```
class no {
   int elem;
   no* prox;
};
```

Em JAVA

```
class no {
   int elem;
   no prox;
};
```

Tipos Ponteiros

- Não se restringe a implementação de tipos recursivos embora seja seu uso principal
- Ponteiro é um conceito de baixo nível relacionado com a arquitetura dos computadores
- O conjunto de valores de um tipo ponteiro são os endereços de memória e o valor nil
- Considerados o goto das estruturas de dados

Tipos Ponteiros

```
#define nil 0
typedef struct no* listaint;
struct no {
   int cabeca;
   listaint cauda;
};
listaint anexa (int cb, listaint cd) {
   listaint l;
   l = (listaint) malloc (sizeof (struct no));
   l->cabeca = cb;
   l->cauda = cd;
   return l;
}
```

Tipos Ponteiros

```
void imprime (listaint l) {
    printf("\nlista: ");
    while (l) {
       printf("%d ",l->cabeca);
       l = l->cauda;
    }
}
main() {
    listaint palindromos, soma10, aux;
    palindromos = anexa(343, anexa(262, anexa(181, nil)));
    soma10 = anexa(1234, palindromos);
    imprime (palindromos);
    imprime (soma10);
```

Tipos Ponteiros

```
    aux = palindromos ->cauda;
    palindromos ->cauda = palindromos ->cauda->cauda;
    free(aux);
    imprime (palindromos);
    imprime (soma10);
}
```

- Atribuição

```
int *p, *q, r;  // dois ponteiros para int e um int
q = &r;         // atribui endereco de r a q
p = q;          // atribui endereco armazenado em q a p
```

- Alocação

```
int* p = (int*) malloc (sizeof(int));
```

Tipos Ponteiros

- Desalocação

```
free(p);
```

- Derreferenciamento implícito

```
INTEGER, POINTER :: PTR
PTR = 10
PTR = PTR + 10
```

- Derreferenciamento explícito

```
int *p;
*p = 10;
*p = *p + 10;
```

Tipos Ponteiros

- Aritmética (C)

```
p++;
++p;
p = p + 1;
p--;
--p;
p = p - 3;
```

- Indexação (C)

```
x = p[3];
```

Ponteiros Genéricos

- Podem apontar para qualquer tipo

```
int f, g;
void* p;
f = 10;
p = &f;
g = *p;        // erro: ilegal derreferenciar ponteiro p/ void
```

- Servem para criação de funções genéricas para gerenciar memória
- Servem para criação de estruturas de dados heterogêneas (aquelas cujos elementos são de tipos distintos)

Problemas Com Ponteiros

- Baixa Legibilidade

  p->cauda = q;

  Inspeção simples não permite determinar qual estrutura está sendo atualizada e qual o efeito

- Possibilitam violar o sistema de tipos

```
int  i, j = 10;
int* p = &j;   // p aponta para a variavel inteira j
p++;           // p pode nao apontar mais para um inteiro
i = *p + 5;    // valor imprevisivel atribuido a i
```

Problemas Com Ponteiros

- Objetos Pendentes

```
int* p = (int*) malloc (10*sizeof(int));
int* q = (int*) malloc (5*sizeof(int));
p = q;                    // area apontada por p torna-se inacessivel
```

  Provoca vazamento de memória

- Referências Pendentes

```
int* p = (int*) malloc(10*sizeof(int));
int* q = p;
free(p);     // q aponta agora para area de memoria desalocada

int* p;
*p = 0;
```

Problemas Com Ponteiros

- Referências Pendentes

```
main() {
    int *p, x;
    x = 10;
    if (x) {
        int i;
        p = &i;
    }
    // p continua apontando para i, que nao existe mais
}
```

Tipo Referência

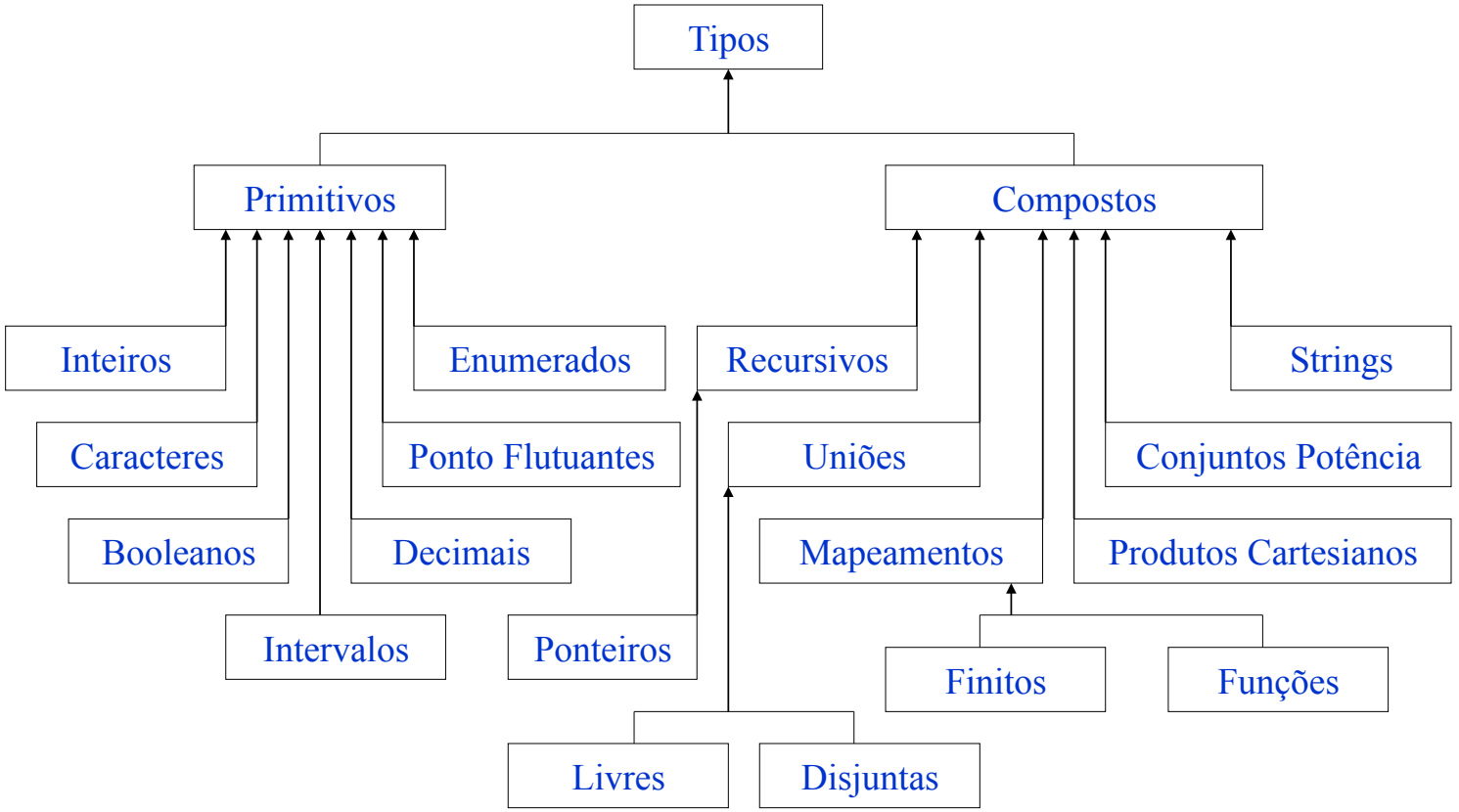
- O conjunto de valores desse tipo é formado pelos endereços das células de memória
- Todas as variáveis que não são de tipos primitivos em JAVA são do tipo referência

```
int res = 0;
int& ref = res;          // ref passa a referenciar res
ref = 100;               // res passa a valer 100
```

Tipo String

- Valores correspondem a uma seqüência de caracteres
- Não existe consenso sobre como devem ser tratadas
- Podem ser consideradas tipos primitivos, mapeamentos finitos ou tipo recursivo lista
- Três formas comuns de implementação
  - Estática (Cobol)
  - Semi-Estática (C, Pascal)
  - Dinâmica (Perl, APL, Snobol)

Hierarquia de Tipos



Conclusões

- Ver os tipos oferecidos por uma LP é um dos primeiros passos no seu estudo;
  - Valores possíveis, como são armazenados, operações disponíveis, etc.
  - Quais tipos podem ser combinados para formar novos tipos;
- Apresentamos uma classificação abstrata, que pode ser usada para estudar qualquer LP.