Linguagens de Programação

Tipos de Dados

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

- Introdução
- Tipos Primitivos
- Tipos Compostos

INTRODUÇÃO

Introdução

- **Valores:** tudo aquilo que pode ser avaliado, armazenado e passado como parâmetro
  - Valores são agrupados em tipos
    - **Ex.:** 3 2.5 'a' "Paulo" 0x1F 026 false

- **Tipos:** conjunto de valores e de operações que estão definidos para os mesmos
  - Podem ser classificados como
    - Primitivos
    - Compostos

Tipos de Dados

- Um **tipo de dados** define uma coleção de dados de objetos e um conjunto de operações predefinidas nestes dados de objetos
- Um descritor é um conjunto de atributos de uma variável
  - Ex.: nome, endereço, valor, tipo, escopo, tempo de vida
- Um objeto representa uma instância de um tipo (dados abstratos) definida pelo usuário

Quais operações são definidas e como elas são especificadas?

TIPOS PRIMITIVOS

Linguagens de Programação

Tipos Primitivos

- Praticamente toda linguagem provê um conjunto de tipos primitivos
- Tipos de dados são definidos em termos de outros tipos ou dele mesmo
- Não podem ser decompostos em valores mais simples, ou seja, são atômicos
- Alguns tipos são meramente reflexos do hardware, outros exigem um pouco de suporte não-hardware para sua implementação

Tipos Primitivos

- Exemplos de tipos primitivos
  - Inteiro
  - Ponto flutuante
  - Complexo
  - Decimal
  - Lógico (**boolean**)
  - Caractere
  - Cadeia de caracteres (**string**)
  - Tipo ordinal (enumerado e intervalo de inteiros)

Tipo Inteiro

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

- Podem existir vários tipos inteiros numa mesma LP, como em C
  - Inteiro com/sem sinal, inteiro base decimal, inteiro base binária, precisão simples, ...

Tipo Inteiro

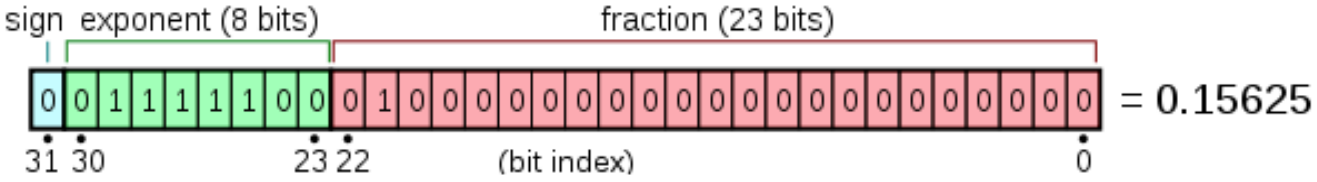
- Tabela com os tipos primitivos inteiros de Java

Tipo	Tamanho (bits)	Intervalo	
		Início	Fim
byte	8	-128	127
short	16	-32,768	32,767
int	32	-2,147,483,648	2,147,483,647
long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

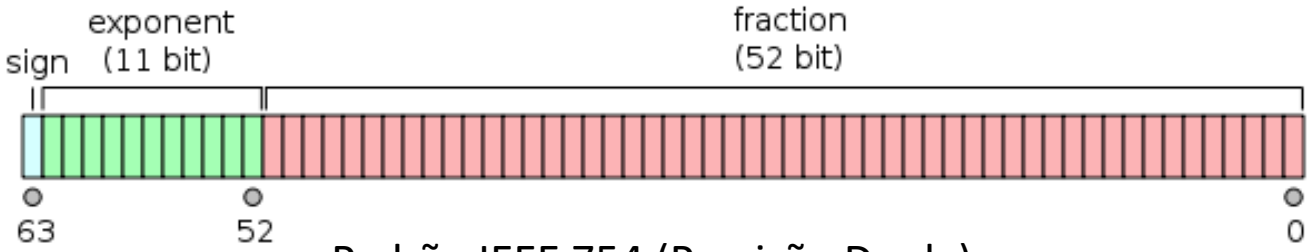
Tipo Ponto Flutuante

- O tipo primitivo ponto flutuante modela os números reais, mas somente como aproximações

- LPs normalmente incluem dois tipos de ponto flutuante
  - *float* (precisão de 32 bits) e *double* (precisão de 64 bits)



Padrão IEEE 754 (Precisão simples)



Padrão IEEE 754 (Precisão Dupla)

Tipo Complexo

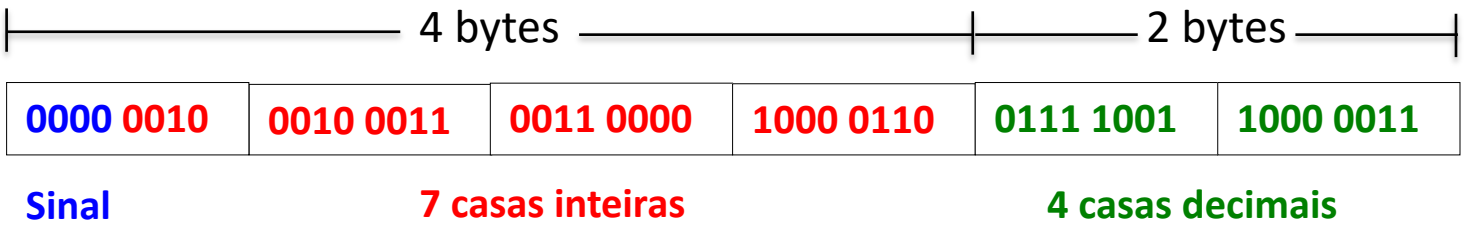
- Algumas linguagens suportam o tipo complexo
  - Ex.: C99, Fortran e Python

- Cada valor consiste de dois números de ponto-flutuante (*float*), a **parte real** e a **parte imaginária**

- Exemplo de número literal em Python: `7 + 3j`
  - `7`: parte real
  - `3j`: parte imaginária

Tipo Decimal

- Armazena um número fixo de dígitos decimais
- Útil para aplicações financeiras
  - Essencial para Cobol
  - Nativo em C#: `decimal salario = 540.0m;`
- Vantagem: acurácia
- Desvantagem: faixa limitada, desperdício de memória
- Exemplo em Cobol (representação interna)



Tipo Lógico (**Boolean**)

- Tipo mais simples
  - Possui apenas dois valores (verdadeiro e falso)
  - Pode ser implementado com 1 bit, mas normalmente é implementado com 1 byte
- Vantagem: Legibilidade
- Exemplos
  - C++ (*bool*), Java (*boolean*)
  - C: não possui o tipo booleano, mas aceita expressões em condicionais
    - **≠ zero** $\Rightarrow$ verdadeiro
    - **= zero** $\Rightarrow$ falso

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 a = 'a' + 3;
printf("%c\n", a);
printf("%d\n", a);
```

```
int d = 65;
printf("%c\n", d);
printf("%d\n", d);
```

DEC	HEX	OCT	CHAR	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH
0	0	000	NUL	32	20	040		64	40	100	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	36	24	044	$	68	44	104	D	100	64	144	d
5	5	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS	40	28	050	(	72	48	110	H	104	68	150	h
9	9	011	TAB	41	29	051	)	73	49	111	I	105	69	151	i
10	A	012	LF	42	2A	052	*	74	4A	112	J	106	6A	152	j
11	B	013	VT	43	2B	053	+	75	4B	113	K	107	6B	153	k
12	C	014	FF	44	2C	054	,	76	4C	114	L	108	6C	154	l
13	D	015	CR	45	2D	055	-	77	4D	115	M	109	6D	155	m
14	E	016	SO	46	2E	056	.	78	4E	116	N	110	6E	156	n
15	F	017	SI	47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020	DLE	48	30	060	0	80	50	120	80	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM)	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC	59	3B	073	;	91	5B	133	[	123	7B	173	{
28	1C	034	FS	60	3C	074	<	92	5C	134	\	124	7C	174	|
29	1D	035	GS	61	3D	075	=	93	5D	135	]	125	7D	175	}
30	1E	036	RS	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

Tipo Cadeia de Caracteres (***String***)

- Valores são sequências de caracteres

- Decisões de projeto
  - Tipo primitivo ou arranjo?
  - Tamanho da *string* estática ou dinâmica?

- Operações com *strings*
  - Atribuição
  - Comparação (=, >, <, etc.)
  - Concatenação (junção ao fim da *string*)
  - Referência a *substring*
  - Correspondência de padrão (*pattern matching*)

Tipo Cadeia de Caracteres (***String***)

- C/C++
  - Não é primitivo
  - Usa arranjos de **`char`** e biblioteca de funções para operações
- SNOBOL4 (LP que manipula *strings*)
  - É primitivo
  - Várias operações, incluindo *pattern matching*
- Java
  - Primitivo através da classe `String`

Tipo Cadeia de Caracteres (***String***)

- Perl
  - Padrões podem ser definidos em termos de expressões regulares
  - Grande poder de expressão
  - Exemplo: palavras contendo unicamente letras

```
if ($dados =~ /[A-Za-z]+/)
    print 'Somente letras';
```

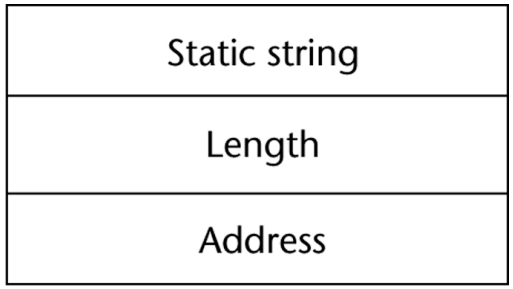
Implementações de ***Strings***

- Estático (COBOL, class *String* de Java)
  - Descritor definido/utilizado em tempo de compilação
- Dinâmico limitado (C e C++)
  - Um caractere especial é usado para indicar o fim da *string*, ao invés de manter o tamanho
- Dinâmico (SNOBOL4, Perl e JavaScript)
  - Precisa de um descritor em tempo de execução
  - Reserva/liberação de memória é um problema

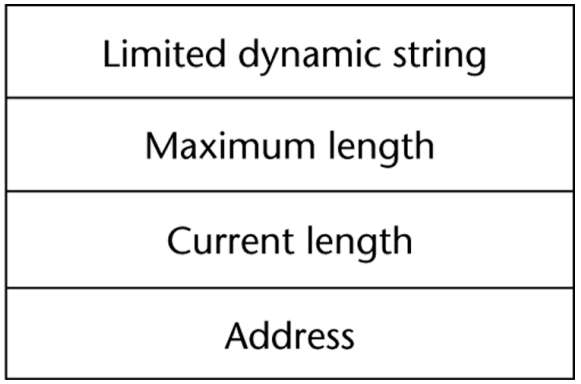
Ada suporta todos os três tipos

Implementações de ***Strings***

- **Tamanho estático:** descritor em tempo de compilação
- **Tamanho dinâmico limitado:** podem precisar de um descritor em tempo de execução (embora não em C/C++)
- **Tamanho dinâmico:** precisa de descritor em tempo de execução; alocação/liberação é o maior problema de implementação



String estática: descritor em tempo de compilação



String dinâmica limitada: descritor em tempo de execução

Tipo Cadeia de Caracteres (***String***)

- Avaliação
  - Ajuda na regibilidade do programa
  - O tipo primitivo de tamanho estático é eficiente, porque não o ter?
  - Tamanho dinâmico é bom, mas muito caro, será que vale a pena?

Tipo Ordinal

- Tipo cuja amplitude de possíveis valores podem ser associados com os inteiros positivos
  - **Ex:** integer, char, boolean

- Muitas linguagens de programação permitem os seguintes tipos ordinais
  - Tipo enumerado
  - Tipo intervalo de inteiros

Tipo Ordinal: Enumerado

- Permite enumerar valores através de constantes simbólicas

- Exemplos
  - Pascal

```
type cor = (vermelho, azul, branco, preto);
```

  - C, C++

```
enum cor { vermelho, azul, branco, preto };
```

  - C#, Java >= 5.0 (Implementado como classe)

```
enum cor { vermelho, azul, branco, preto };
```

Tipo Ordinal: Enumerado

- Considerações de projeto
  - Pode uma constante simbólica pertencer a mais de uma definição de tipo? Se sim, como verificar?
  - As enumerações podem ser convertidas em inteiros?
  - Algum outro tipo pode ser convertido para uma enumeração?

Tipo Ordinal: Enumerado

- Vantagens
  - Legibilidade
    - **Ex:** não precisa codificar cores como inteiros
  - Confiabilidade
    - Não permite que se opere cores (soma)
    - Não se pode definir valores fora da faixa da enumeração
    - Ada, C# e Java >= 5.0 não fazem coerção para inteiros

Tipo Ordinal: Intervalo de Inteiros

- Subsequência ordenada contínua de um tipo enumerado ordinal

- Exemplo
  - Pascal

```
type positivo = 0 .. MAXINT;
```

  - ADA

```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
```

Tipo Ordinal: Intervalo de Inteiros

- Avaliação
  - Melhora legibilidade
  - Podem guardar apenas certos valores limitados à faixa

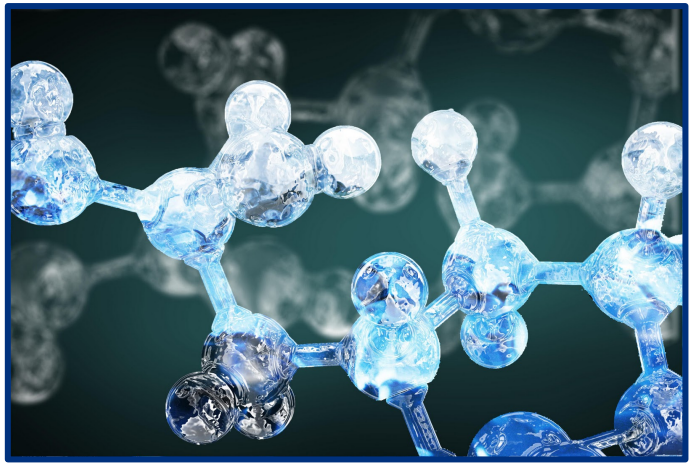
- Melhora confiabilidade
  - Detecção de erros e das amplitudes dos valores

Tipo Ordinal: Intervalo de Inteiros

- Implementação
  - Tipos enumerados geralmente são implementados associando números inteiros a cada constante
  - Subfaixas são implementadas da mesma forma que os tipos pais

- O código para restringir atribuições a variáveis de subfaixas deve ser inserido pelo compilador

TIPOS COMPOSTOS

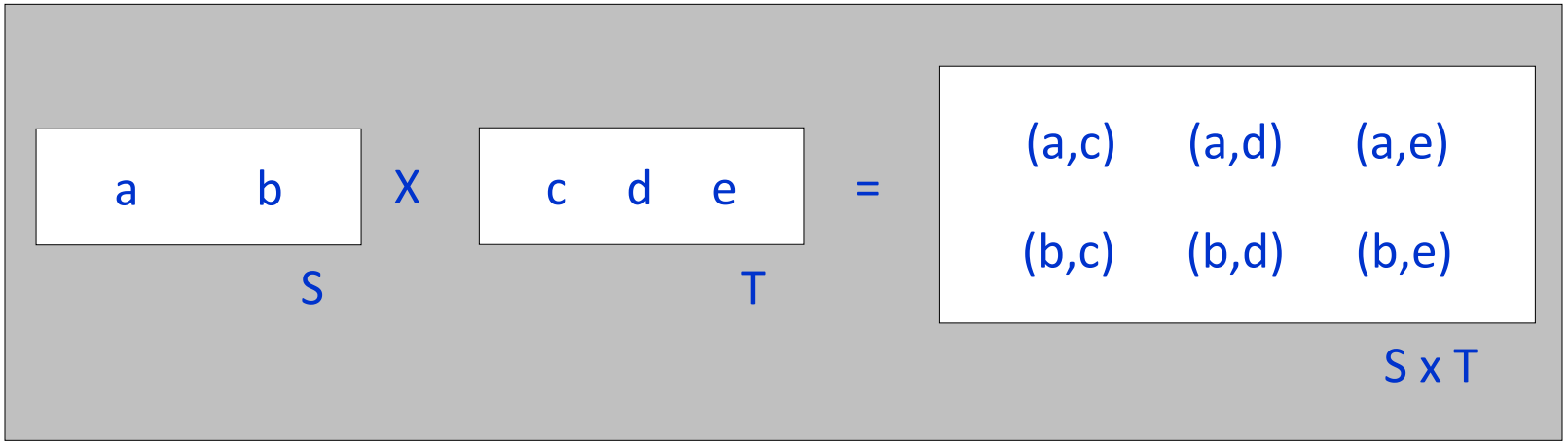


Tipos Compostos

- Aqueles que podem ser criados a partir de tipos mais simples
  - Registros, vetores, listas, arquivos
- Entendidos em termos dos conceitos
  - Produto cartesiano, uniões (livres e disjuntas), 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



- Cardinalidade

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

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;
```

- Em LPs orientadas a objetos, produtos cartesianos são definidos a partir do conceito de classe (Java só tem classes)

Produto Cartesiano

- Exemplo

```
struct data {
    int dia, mes, ano;
};
```

  - Inicialização

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

  - Acesso aos membros (através de seletores)

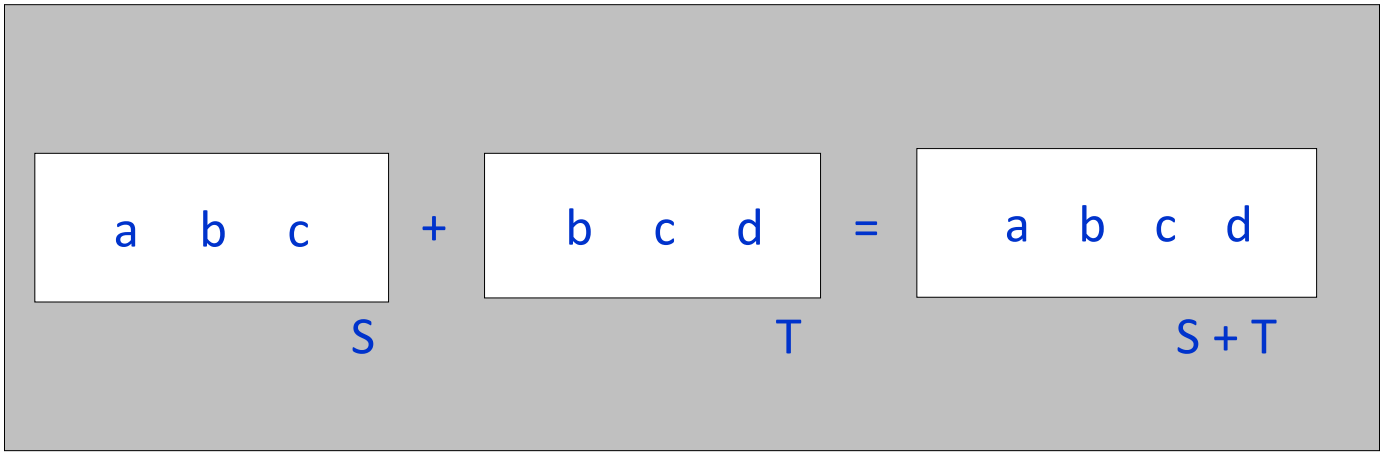
```
printf("%02d\n", d.dia);
d.mes = 10;
```

  - Cardinalidade

```
#INTEGER x #INTEGER x #INTEGER
```

Uniões

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



- Cardinalidade

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

Uniões

- Uniões livres
  - Pode haver interseção entre o conjunto de valores dos tipos que formam a união
  - Há possibilidade de violação no sistema de tipos

- Exemplo

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

```
void funct() {
    union medida medicao;
    float altura;

    medicao.centimetros = 180;
    altura = medicao.metros;
    printf("\naltura: %f m\n", altura);
}
```

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

- Exemplo em Pascal

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

```
var num of Numero;
Numero.tag = decimal;
Numero.val = 1.5;
```

Uniões

- Exemplo

```
TYPE TipoProduto = (musica, livro, video);
Compra = RECORD
    valor: REAL;
    CASE produto: TipoProduto OF
        musica: (numeromusicas: INTEGER);
        livro: (numeropaginas: INTEGER);
        video: (duracao: INTEGER, colorido: BOOLEAN);
END;
```

  - Possíveis valores

```
(25.00, musica(16))
(35.00, livro(257))
(40.00, video(121, TRUE))
```

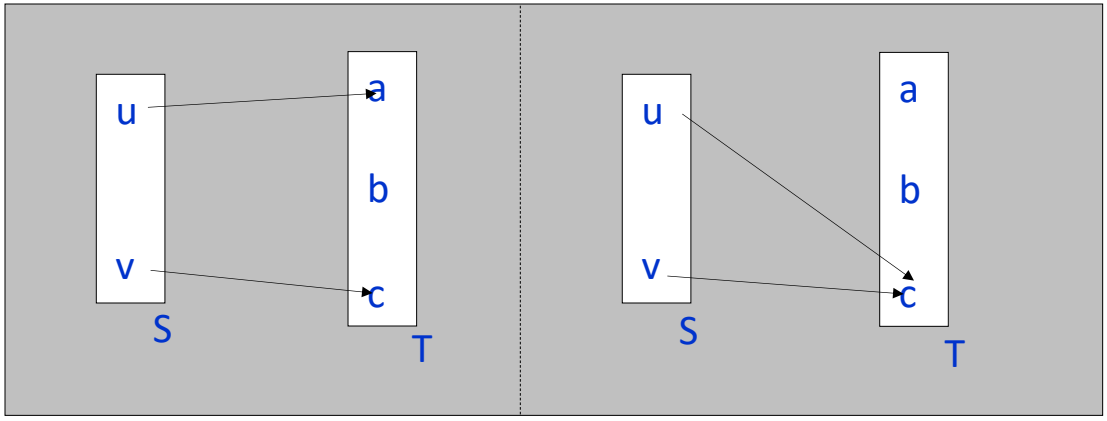
  - Cardinalidade

```
#REAL x (#INTEGER + #INTEGER + (#INTEGER x #BOOLEAN))
```

Mapeamentos

- Mapeamentos podem ser
  - Finitos
  - Através de funções

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

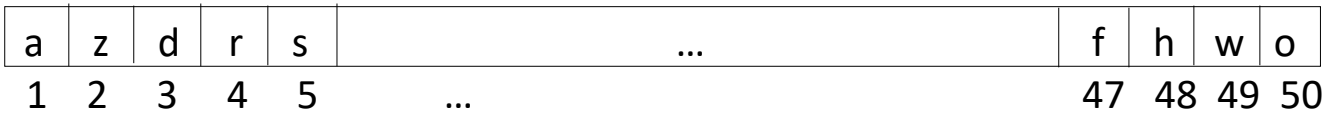


- Cardinalidade

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

Mapeamentos Finitos

- O conjunto domínio é finito
- Vetores e matrizes
  - **array** S **of** T; **(S → T)**
  - **array** [1..50] **of** char; **([1,50] → char)**



- O conjunto índice deve ser finito e discreto

- Verificação de índices em C vs. JAVA: verificação em tempo de execução aumenta a confiabilidade, mas perde eficiência

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

Categoria de Vetores

- Estáticos (em C)

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

- Semi-dinâmicos (em C ISO/99)

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

- Semi-estáticos (em C)

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

- Dinâmicos (em C++)

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

Categoria de Vetores

- Tabela com as categorias de vetores

Categoria de Vetor	Tamanho	Tempo de Definição	Alocação	Local de Alocação	Exemplos de LPs
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	Heap	APL, PERL

Vetores Dinâmicos

- Podem ser implementados em C, C++ e JAVA através do monte (heap)
- 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 liberação também. Isso torna a programação mais complexa e suscetível a erros

Vetores Multidimensionais

- Exemplo

```
int mat[5][4];
```

  - Representação

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

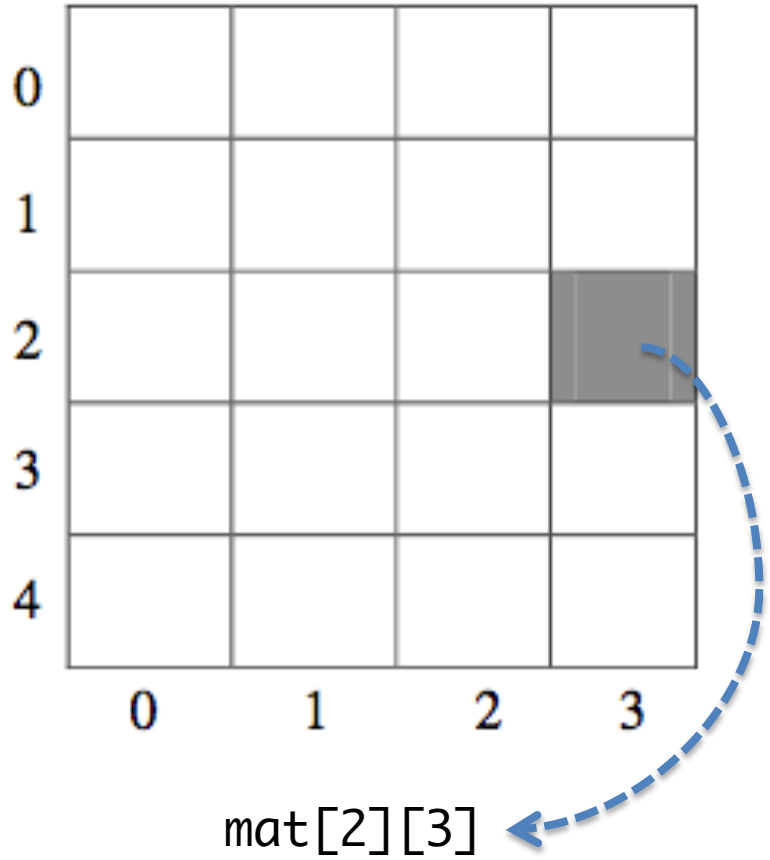
  - Conjunto de valores

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

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

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

    $(\#INT)^{20}$



Vetores Multidimensionais

- Elementos são acessados através da aplicação de fórmulas

```
posição mat[i][j] = endereço de mat[0][0] +
                        i x tamanho da linha +
                        j x tamanho do elemento
                  = endereço de mat[0][0] +
                        (i x número de colunas + j) x
                        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

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

Operações com Vetores

- Vetores podem suportar as seguintes operações
  - Indexação
  - Inicialização
  - Atribuição
  - Comparação (igualdade e desigualdade)
  - Concatenação

Mapeamentos Através de Funções

- Uma função implementa um mapeamento S → T através de um algoritmo
- O conjunto S não necessariamente é 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 exemplos: `palindromo`, `impar`, `par`, `primo`

Mapeamentos Através de Funções

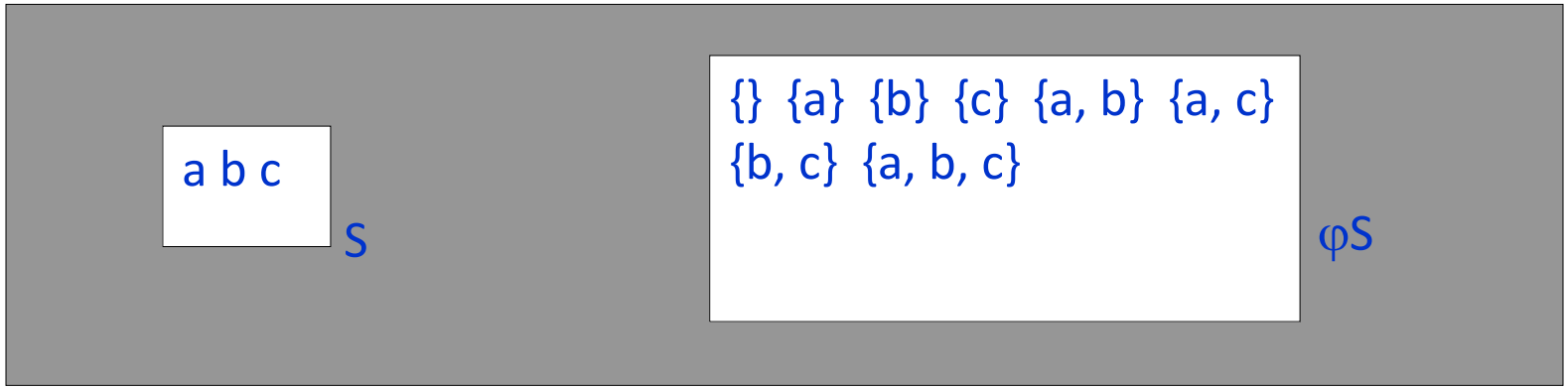
- C utiliza o conceito de ponteiros para manipular endereços de funções como valores

```
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;
}
void 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 como valores

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



- Cardinalidade

  $\#\varphi S = 2^{\#S}$

Conjuntos Potência

- Operações básicas
  - Pertinência
  - Contém
  - Está contido
  - União
  - Diferença
  - Diferença simétrica
  - Interseção

Conjuntos Potência

- Poucas linguagens de programação oferecem o tipo conjunto potência, muitas vezes de forma restrita

- Exemplo em Pascal

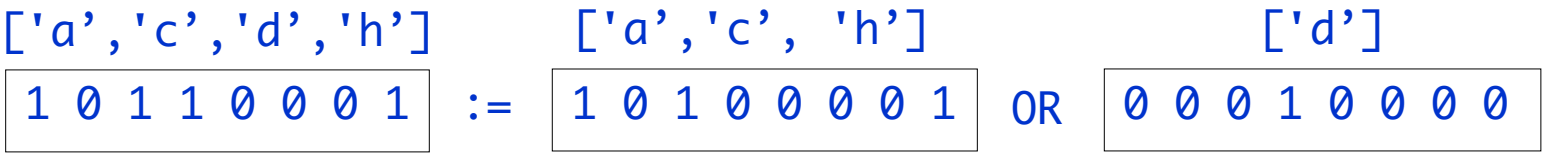
```
TYPE Carros = (corsa, palio, gol);
    ConjuntoCarros = SET OF Carros;
VAR Carro: Carros;
    CarrosPequenos: ConjuntoCarros;
BEGIN
    Carro:= corsa;
    CarrosPequenos := [palio, gol];              /*atribuicao*/
    CarrosPequenos:= CarrosPequenos + [corsa];   /*uniao*/
    CarrosPequenos:= CarrosPequenos * [gol];     /*intersecao*/
   IF (Carro IN CarrosPequenos) THEN ...         /*pertinencia*/
   IF (CarrosPequenos >= [gol, corsa]) THEN ... /*contem*/
```

Conjuntos Potência

- Restrições de PASCAL visam permitir implementação eficiente, através de mapas de bits

- Exemplo

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



Recursivos

- Tipos recursivos são tipos de dados cujos valores são compostos por valores do mesmo tipo
  - R ::= <parte inicial> R <parte final>
  - Lista ::= Lista Vazia | (Elemento x 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

Recursivos

- Tipos recursivos podem ser definidos a partir de ponteiros ou através de referências

- Exemplos

```c
struct No {
    int  elem;
    struct No* prox;
};
```

Em C

```cpp
class No {
public:
   int elem;
   No* prox;
};
```

Em C++

```java
class No {
   int elem;
   No prox;
};
```

Em Java

Ponteiros

- Não se restringe a implementação de tipos recursivos embora seja um de seus usos principais
- 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**

Ponteiros

- Atribuição

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

- Alocação

```
int* p = (int*) malloc(sizeof(int)); // em C
int* p = new int; // em C++
```

- Liberação

```
free(p); // em C
delete p; // Em C++
```

Ponteiros

- Dereferenciamento implícito (FORTRAN 90)

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

- Dereferenciamento explícito (C)

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

Ponteiros

- Aritmética de ponteiros

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

- Indexação de ponteiros

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

Ponteiros Genéricos

- Aritmética de ponteiros

```
int f, g;
void* p;
f = 10;
p = &f;
g = *p; // erro: é ilegal dereferenciar 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: inspeção simples não permite determinar qual estrutura está sendo atualizada e qual o efeito

```
p->cauda = q;
```

- 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
```

- Objetos pendentes: provoca vazamento de memória

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

Problemas com Ponteiros

- Referências pendentes
  - Exemplo 1

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

  - Exemplo 2

```
int *r;    // ponteiro não inicializado
*r = 0;
```

  - Exemplo 3

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

Referência

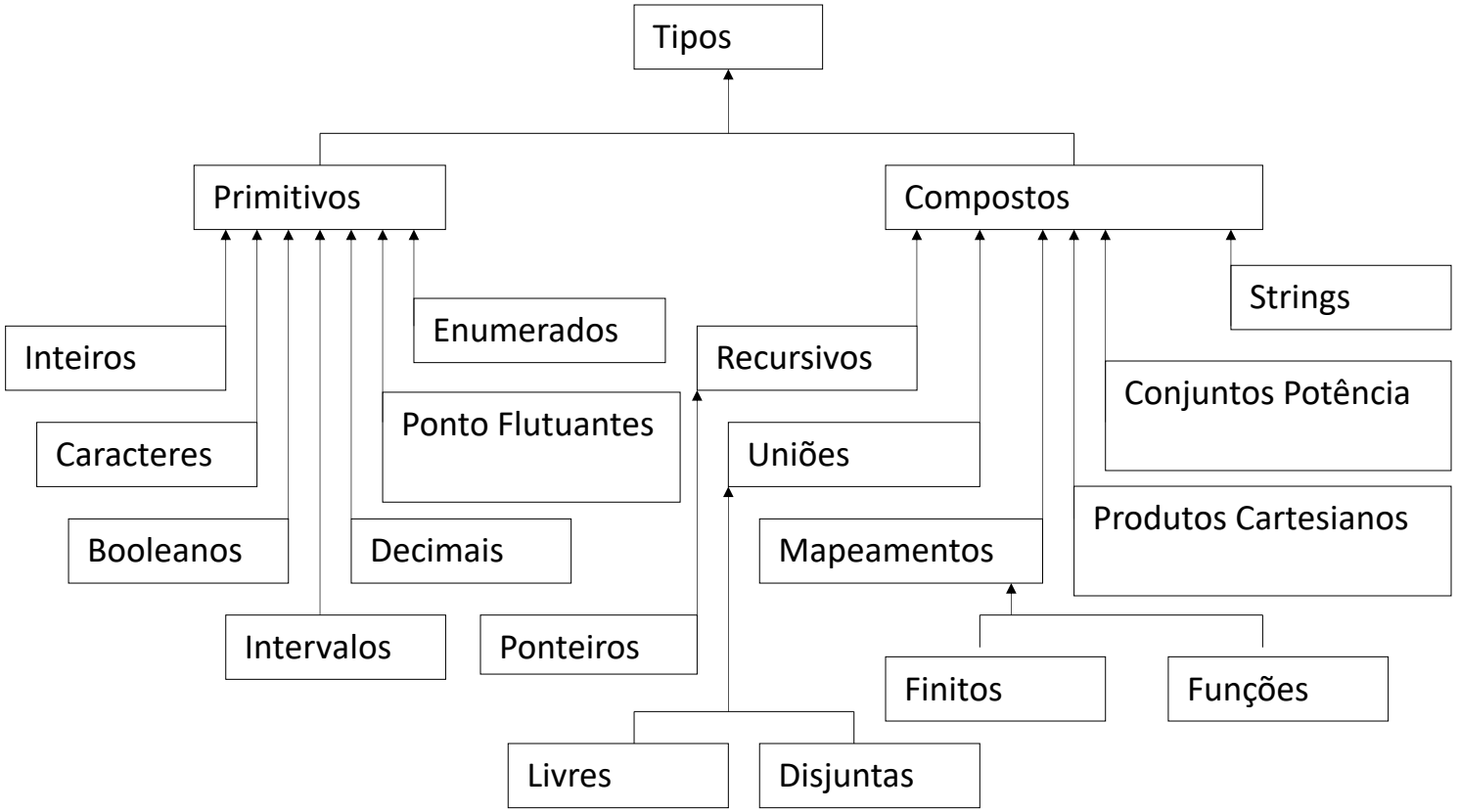
- O conjunto de valores desse tipo é formado pelos endereços das células de memória

- Todas as variáveis que não são de tipos primitivos em JAVA são do tipo referência

- Exemplo (em C++)

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

Hierarquia de Tipos



ISSO É TUDO PESSOAL!