**Linguagens de Programação**

# Tipos

Andrei Rimsa Álvares
andrei@cefetmg.br




# Sumário

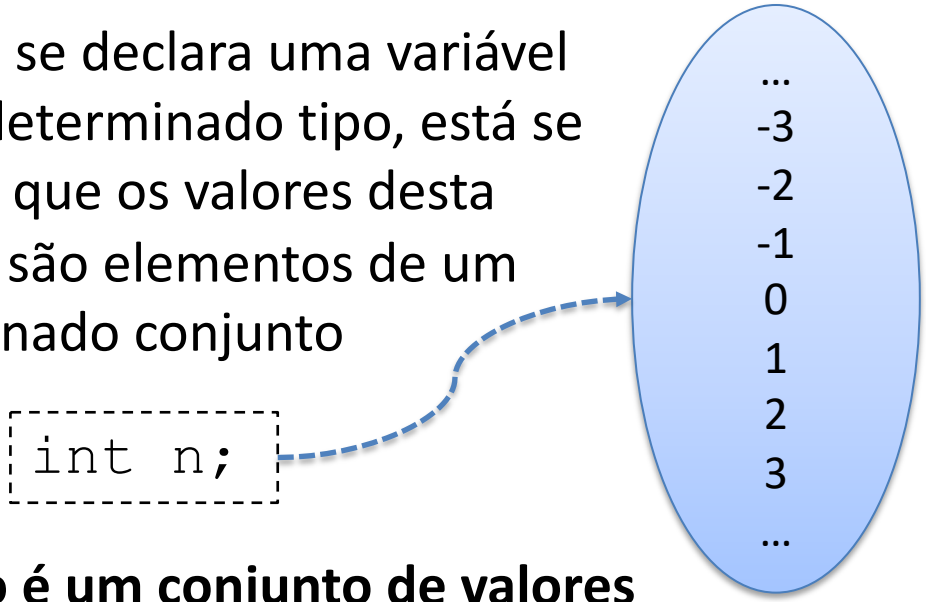
- Introdução
- Miscelânea de tipos
  - Tipos primitivos
  - Tipos construídos
- Usos para tipos
  - Anotação de tipos
  - Inferência de tipos
  - Verificação de tipos
  - Equivalência de tipos

# INTRODUÇÃO


**Linguagens de Programação**

# Introdução

- Quando se declara uma variável de um determinado tipo, está se dizendo que os valores desta variável são elementos de um determinado conjunto




**Cuidado:** A variável `n` não pode receber a string `"olá"` ou o real `3.14`

Quantos elementos tem este conjunto? Quais são seus limites?

- **Um tipo é um conjunto de valores**
  - Com uma representação em baixo nível
    - Os elementos do conjunto compartilham uma representação comum: são representados do mesmo jeito na memória
  - Com uma coleção de operações aplicadas sobre eles
    - Exemplo: subtração para qualquer par de valores de `int`

# MISCELÂNIA DE TIPOS

# Miscelânea de Tipos

- Existem muitos tipos, não é possível cobrir todos
  - Será feito um tour em apenas alguns deles
- A forma de construir conjuntos na matemática é parecida com a forma de construir tipos em linguagens de programação
  - Será visto como esta conexão é feita

Um tipo é um conjunto

# Tipos Primitivos vs. Construídos

- Todo sistema de tipos começa com **tipos primitivos**
  - Por exemplo, os tipos `int` e `real` de ML

- As linguagens modernas permitem definir tipos construídos em termos de tipos primitivos
  - Por exemplo, em ML pode-se definir o tipo par de inteiros

```
type intpair = int * int;
```

*"Qualquer tipo que um programa pode usar, mas não definir por si próprio é um tipo primitivo; qualquer tipo que um programa pode definir por si próprio (usando tipos primitivos) é um tipo construído."*

# O tipo `bool`

- Considere os seguintes trechos de códigos que usam o tipo `bool`

```
#include <stdbool.h>
int main() {
  bool b = true;
  return b ? 1 : 0;
}
```

Em C

```
val b : bool = true;
val x = if b then 1
            else 0;
```

Em ML

```
b = True
print(1 if b else 0)
```

Em Python

O tipo `bool` nestas linguagens é primitivo ou construído?

# O tipo `bool`

- Não é primitivo em C, pois pode ser definido

```
typedef enum {
    false,
    true
} bool;
```

- Não é primitivo em ML, pois pode ser definido

```
datatype bool = true | false;
```

- É primitivo em Python

Como saber se um tipo é primitivo ou construído?

# MISCELÂNIA DE TIPOS > TIPOS PRIMITIVOS

# Tipos Primitivos

- A definição de cada linguagem de programação descreve quais são seus tipos primitivos

- Algumas linguagens usam uma definição mais estrita dos tipos primitivos que outras
  - Algumas definem os tipos primitivos de forma exata
    - Ex.: Java
  - Outras deixam a escolha mais livre para o implementador da linguagem
    - Ex.: C e ML

# Tipo Inteiro

- Em JavaScript, têm-se inteiros de qualquer tamanho (*Number* vs. *BigInt*)
- Em Java, o tamanho do tipo inteiro é fixado pela linguagem

| 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 |

- Em C, o tamanho é variado dependendo da sua arquitetura
  - `short int`, `unsigned short int`, `int`, `unsigned int`, `long int`, `unsigned long int`

Como saber a faixa do tipo `int` para sua arquitetura em C e ML?

# Tipo Inteiro

- Em ML têm-se as constantes `Int.minInt` e `Int.maxInt`

```
- Int.minInt;
val it = SOME ~4611686018427387904 : int option
- Int.maxInt;
val it = SOME 4611686018427387903 : int option
```

- Em C, têm-se as constantes `INT_MIN` e `INT_MAX` definidas no cabeçalho `<limits.h>`

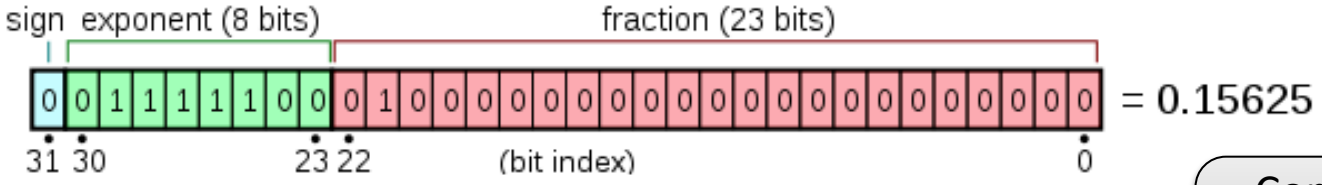
```
#include <stdio.h>
#include <limits.h>

void main() {
  printf("%d\n", INT_MIN);
  printf("%d\n", INT_MAX);
}
```

É possível obter estes limites sem o auxílio destas constantes?

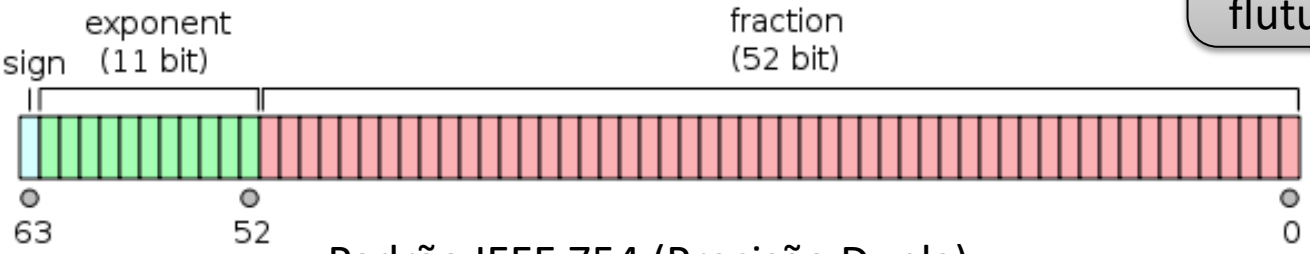
# Tipo Ponto Flutuante

- 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)

Como é feito o cálculo do número em ponto flutuante neste padrão?

# Tipo Ponto Flutuante

- Considere o seguinte programa em C para cálculo do valor em centavos de uma determinada quantidade em reais

```
#include <stdio.h>

void main() {
  printf("Valor em reais: ");
  float reais;
  scanf("%f", &reais);
  int cents = reais * 100;
  printf("Valor em centavos: %d\n", cents);
}
```

  - O valor de R$5,50 equivale a 550 centavos

```
$ ./conversao
Valor em reais: 5.50
Valor em centavos: 550
```

E qual o valor em centavos de R$4,18?



# 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 complexo em Python

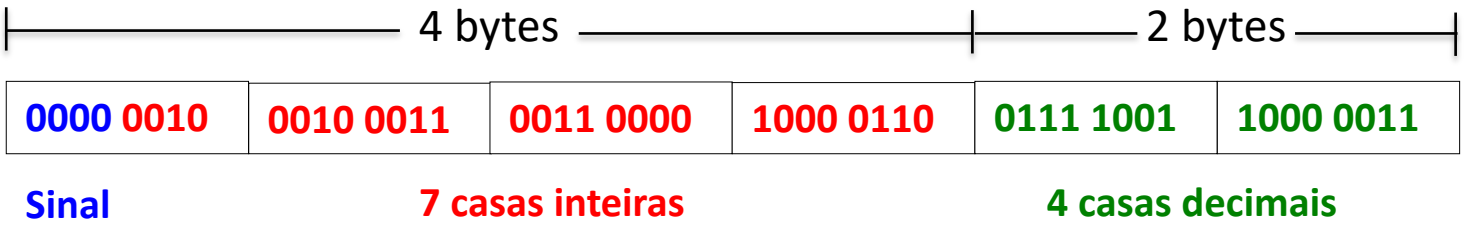
```
$ python3
>>> 7 + 3j
(7+3j)
>>> (5 + 2j) * 3
(15+6j)
```

# Tipo Decimal

- Armazena um número fixo de dígitos decimais
- Útil para aplicações financeiras
  - Essencial em 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 primitivo, 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);
```

♣: https://symbl.cc/en/unicode-table

| 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
  - 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
  - Possui grande poder de expressão
  - Exemplo de palavras contendo unicamente letras

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

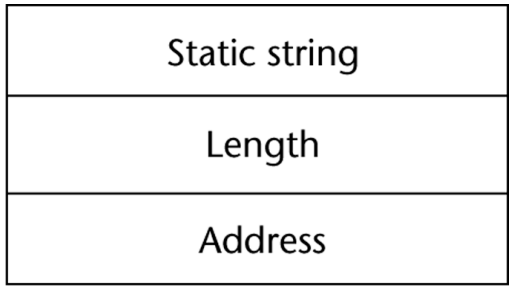
# Implementações de ***Strings***

- Estático (COBOL,  a classe *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

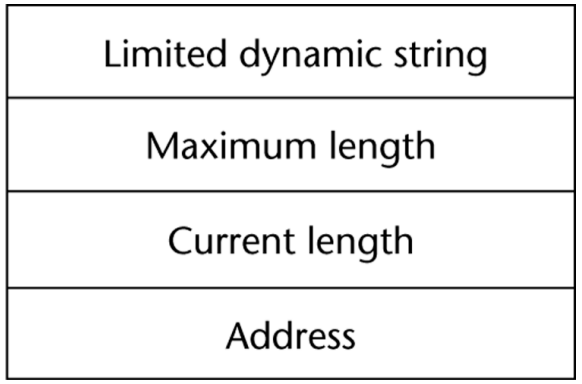
Ada suporta todos os três tipos

# 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

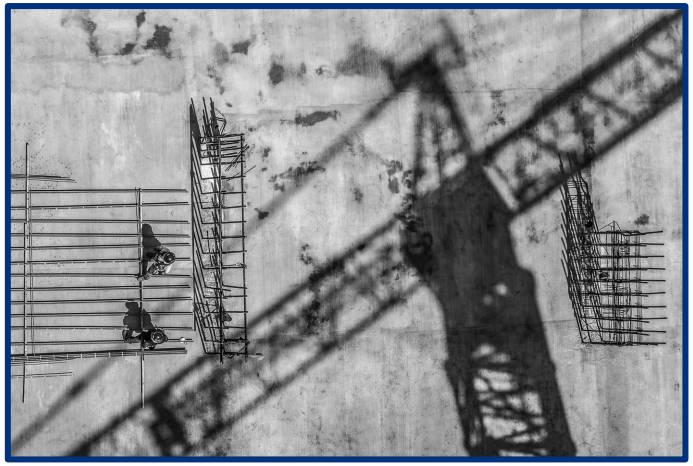



**String estática:** descritor em tempo de compilação




**String dinâmica limitada:** descritor em tempo de execução

# MISCELÂNIA DE TIPOS > TIPOS CONSTRUÍDOS



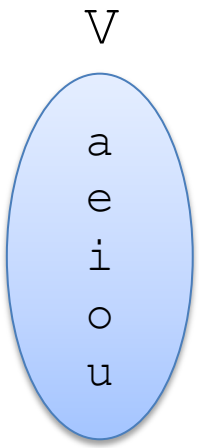
Linguagens de Programação

# Tipos Construídos

- Como linguagens geralmente possuem uma quantidade limitada de tipos primitivos, um sistema de tipos extenso e expressivo deve suportar tipos construídos
  - Como enumerações, tuplas, arranjos, listas, uniões, subtipos e funções

- A maioria das linguagens modernas possuem construtores de tipos expressivos o suficiente para construir infinitos tipos

# Enumerações




- Na matemática, um conjunto é construído listando todos os seus elementos
  - Por exemplo, o conjunto de todas as vogais minúsculas
    `V = { a, e, i, o, u }`

- Similarmente, muitas linguagens permitem criar um tipo enumerado listando todos os seus elementos
  - Em C: `enum coin { penny, nickel, dime, quarter };`
  - Em ADA: `type GENDER is ( MALE, FEMALE );`
  - Em Pascal: `type primaryColors = ( red, green, blue );`
  - Em ML: `datatype day = M | Tu | W | Th | F | Sa | Su;`

Enumerações permitem definir uma coleção (ex: `coin`) de constantes nomeadas (ex.: `nickel`)

Como estas constantes são representadas internamente?

# Enumerações

- A forma mais comum de representação é associar um inteiro a cada elemento: `penny=0`, `nickel=1`, `dime=2`, ...

Qual o resultado de `penny+nickel`?

- Algumas linguagens expõem esta representação aos programadores
  - Em pascal enumerações podem ser usadas como índices em laços

```
for C := red to blue do P(C)
```

  - Em C este conceito é elevado ao extremo, tal que as constantes são tratadas literalmente como inteiros

```
enum coin { penny=1, nickel=5, dime=10, quarter=25 };
```

- Outras linguagens escondem completamente a sua implementação
  - Em ML, somente é possível comparar as constantes por igualdade

```
fun isweekend x = (x = Sa orelse x= Su);
```

Como funciona em Java?

# Tuplas

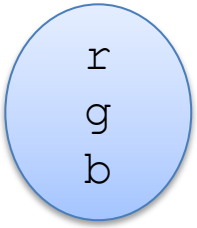
- O produto cartesiano de dois conjuntos A e B ($A\ x\ B$) é o conjunto de todos os pares formados pelos elementos do primeiro com o segundo

$$S = A\ x\ B$$
$$= \{\ (a,b)\ |\ a \in A\ \wedge\ b \in B\ \}$$

A

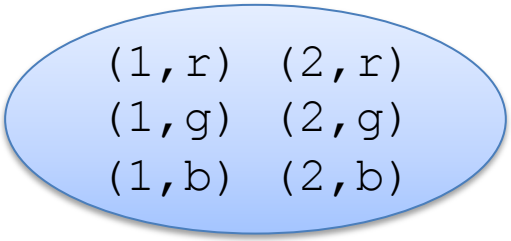

```
A = {
   1, 2
}
```

B




```
B = {
   r, g, b
}
```

AxB




```
A x B = {
   (1,r), (2,r),
   (1,g), (2,g),
   (1,b), (2,b)
}
```

# Tuplas

- Algumas linguagens, como ML, possuem o tipo tupla diretamente

```
type irpair = int * real;
```

O que acontece se aplicar a mais conjuntos?

- Outras linguagens suportam tuplas nomeadas (estruturas ou registros)

```
struct complex {
  double rp;
  double ip;
};
```

Em C

```
type complex = {
  rp : real,
  ip : real
};
```

Em ML

Como representar tuplas na memória?

# Tuplas

- ML esconde a implementação de tuplas do programador
  - Além de comparações, a única outra operação suportada é extração de seus elementos

```
- fun getFirstPart (x : irpair) = #1 x;
val getFirstPart = fn : irpair -> int
```

```
- fun getip (x : complex) = #ip x;
val getip = fn : complex -> real
```

- C cuidadosamente define quais partes da representação devem ser a mesma para qualquer implementação e quais partes podem diferir
  - Os elementos devem aparecer na memória na mesma ordem que aparecem no registro (exceto para campos de bits)
  - O primeiro elemento deve estar no começo sem buracos antes dele
    - Já os outros buracos são dependentes da implementação

# Conjunto de Vetores

- $A^n$ é o conjunto de todos os vetores de tamanho `n` que possuem elementos de `A` em todas as posições (vetores de tamanho fixo)

$$S = A^n$$
$$= A\ x\ A\ x\ \ldots\ x\ A\ (n\ vezes)$$
$$= \{\ (a_1, \ldots, a_n) \mid \forall a_i \in A, 1 \leq i \leq n\ \}$$

É equivalente a fazer o produto cartesiano de `A` `n` vezes

- $A^*$ é o conjunto de todos os vetores de qualquer tamanho que possuem elementos de `A` em todas as posições (vetores de tamanho variado)

$$S = A^*$$
$$= \bigcup_i A^i$$

O conceito de ambos os tipos de conjuntos podem ser aplicados a arranjos, listas e strings

Como indexar os elementos de um vetor?

# Valores de Índices

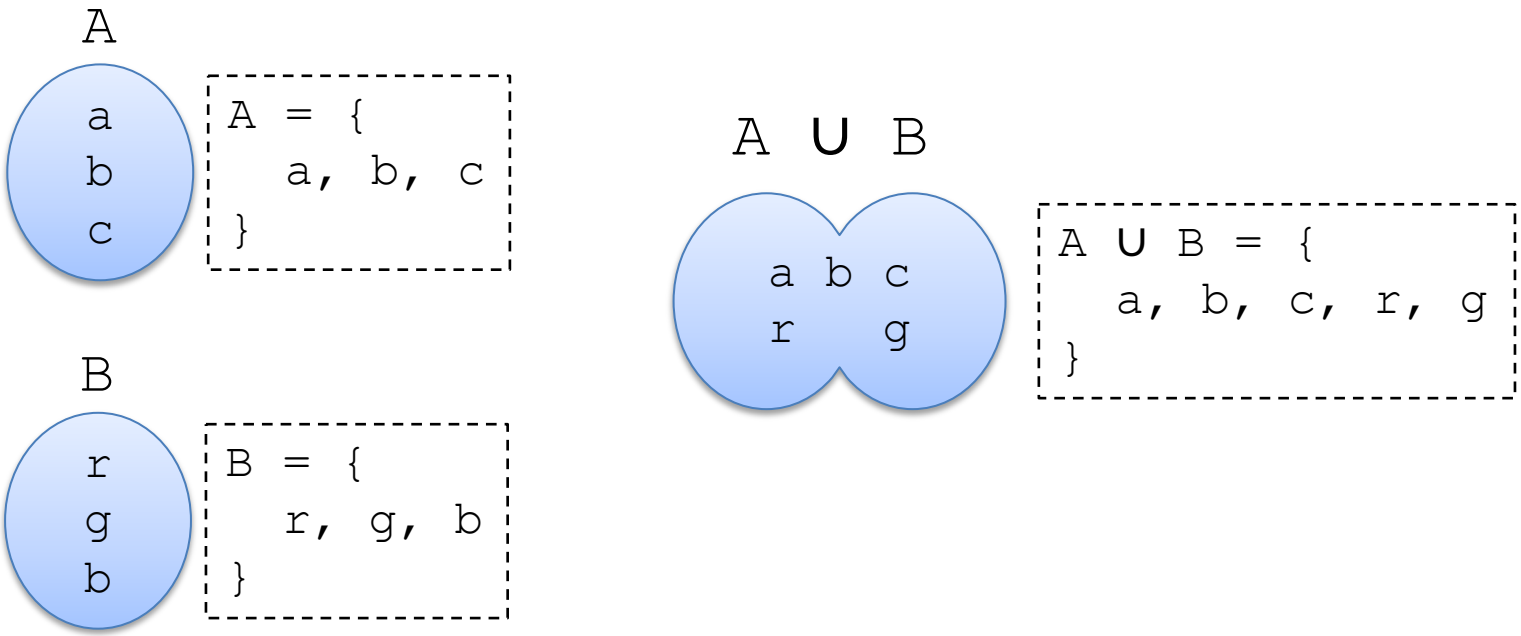
- Algumas linguagens como C, C++ e Java usam índices fixos
  - O primeiro elemento de um arranjo `a` é `a[0]` e o último é `a[n-1]`, tal que `n` é o tamanho do arranjo
- Outras linguagens como Pascal são mais flexíveis
  - Índices podem ser inteiros, caracteres, enumerações ou intervalos
  - O índice inicial e final (fixo em tempo de compilação) pode ser escolhido pelo programador

```
type
    LetterCount = array ['a'..'z'] of Integer;
var
    Counts: LetterCount;
begin
    Counts['a'] = 1
    ...
```

# Uniões

- A união de dois conjuntos A e B ($A \cup B$) é o conjunto de todos os elementos que estão em A ou em B (ou em ambos)

$$S = A \cup B$$
$$= \{ x \mid x \in A \vee x \in B \}$$



# Uniões

- A união de dois tipos é uma ideia natural que aparece em muitas linguagens de programação

```
union element {
  int i;
  double d;
};
```

Em C

```
datatype element =
  I of int | R of real;
```

Em ML

Qual a diferença de `type` para `datatype`?

Como representar uniões na memória?

# Uniões

- C usa uma representação mais solta para uniões
  - A representação de inteiros se for um `int` e a representação de ponto flutuante se for um `double;` a união tem memória suficiente para qualquer um dos dois, o que for maior

```
union element e;
// sizeof(e) = max(sizeof(e.i), sizeof(e.d));
```

  - Porém, não é possível saber somente olhando o valor do tipo `element` se deveria ser um número inteiro ou ponto flutuante

```
e.i = 100;
double x = e.d;
```

Qual valor é impresso para a variável `x`?

# Uniões

- ML usa uma representação mais estrita para uniões que permite pegar mais erros de tipos em tempo de compilação
  - Só é possível extrair seu conteúdo e você deve especificar o que fazer para cada tipo da união

```
- fun getReal (R x) = x
=   | getReal (I x) = real x;
val getReal = fn : element -> real
```

O que acontece se você omitir um caso?

# Uniões

- ADA e Modula-2 usam uma abordagem mista (**registros variantes**)
  - A ideia é suportar uniões em um registro com uma enumeração: o valor do tipo enumerado determina o tipo interno da união

```
type DEVICE is (PRINTER, DISK);

type PERIPHERAL(Unit: DEVICE) is
  record
    HoursWorking: INTEGER;
    case Unit is
      when PRINTER =>
        Line_count: INTEGER;
      when DISK =>
        Cylinder: INTEGER;
        Track: INTEGER;
    end case;
  end record;
```

Em ADA

# Subtipos

- Subtipos é formado por elementos de um subconjunto de elementos de algum tipo existente $S = \{ x \in X \mid P(x) \}$
- Linguagens suportam subtipos de forma mais ou menos geral
  - Menos geral: subfaixas de pascal

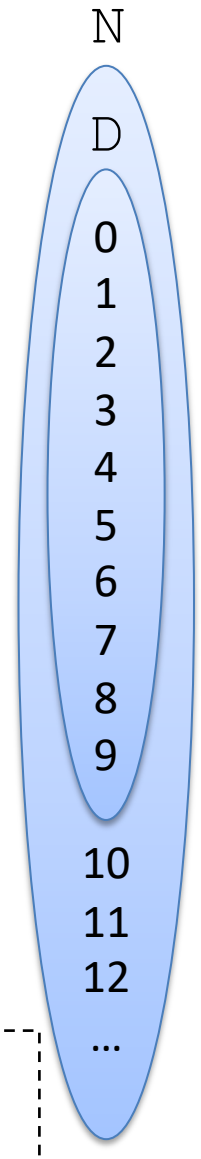
```
type digit = 0..9;
```

  - Médio geral: subtipos de ADA

```
subtype DIGIT is INTEGER range 0..9;
subtype WEEKDAY is DAY range MON..FRI;
subtype DISK_UNIT is PERIPHERAL(DISK);
```

  - Mais geral: tipos de Lisp com predicados

```
(declare (type integer x))
(declare (type (or null cons) x))
(declare (type (and number (not integer)) x))
(declare (type (and integer (satisfies evenp)) x))
```



# Subtipos

- Normalmente subtipos suportam todas as mesmas operações que são suportadas pelo seu supertipo

- Outras operações podem ser adicionadas ao subtipo que não fazem sentido para o supertipo

```
function toDigit(X: Digit): Char;
```

Um subtipo é um subconjunto de valores, mas ele pode suportar um superconjunto de operações

# Funções

- Uma função ($f : A \rightarrow B$) é um mapeamento dos elementos de um conjunto $A$ (domínio) em elementos de um conjunto $B$ (imagem)

$$S = \{ f \mid dom\ f = A \wedge img\ f = B \}$$

- A maioria das linguagem tem uma ideia similar sobre tipo de funções

```
int f(char a, char b) {
  return a == b;
}
```

Em C

Quais operações podem ser feitas com funções?

```
fun f (a : char, b : char) = (a = b);
```

Em ML

# USOS PARA TIPOS

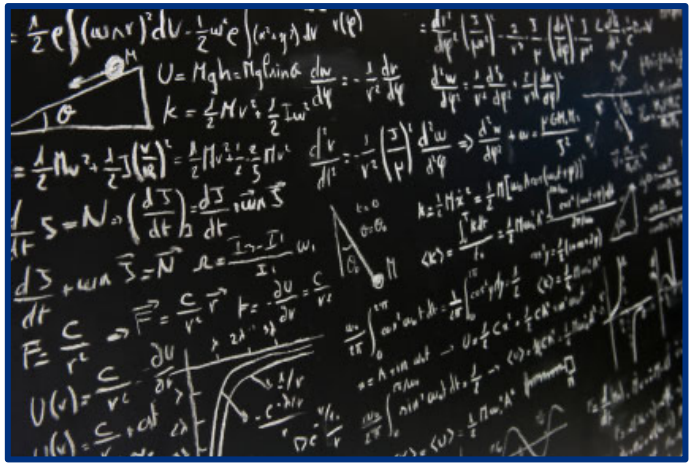


# Tipos

- Tipos são usados de diversas formas diferentes por linguagens de programação

- Vamos ver sobre
  - Anotações de tipos
  - Inferência de tipos
  - Verificação de tipos
  - Equivalência de tipos

# USOS PARA TIPOS > ANOTAÇÃO DE TIPOS



# Anotação de Tipos

- Várias linguagens exigem, ou pelo menos, permitem anotações de tipos para variáveis, funções, ...
  - Java exige anotações de tipos em todas as definições de variáveis
  - ML permite colocar anotações de tipos, não só em variáveis/funções, mas também em expressões
  - Somente Prolog não possui nenhum sistema de anotação de tipos
- O programador usa anotações de tipo para fornecer informações de tipos estáticas para o sistema da linguagem
  - São uma forma de documentação e programas são mais legíveis
- Parte da linguagem é sintaxe para descrever tipos
  - Considere `*`, `->` e `list` em ML

# Anotação de Tipos

- Algumas linguagens usam tipos intrínsecos
  - Em dialetos de BASIC: `S` é um número, enquanto `S$` é uma string
  - Em dialetos de Fortran: variáveis que começam com `I`, `J`, `K`, `L`, `M`, ou `N` são `INTEGER`, caso contrário são `REAL`
  - Em Perl: variáveis que começam com `$` são escalar, se com `@` são arranjo (`array`) e se com `%` são dicionário (`hashtable`)

# USOS PARA TIPOS > INFERÊNCIA DE TIPOS

# Inferência de Tipos

- Sistemas de linguagens obtêm informações de tipos também de outras fontes
  - Constantes possuem tipo
    - Em Java, a constante `10` possui tipo `int` e `10l` possui tipo `long`; já `1.0` possui tipo `double` e `1.0f` o tipo `float`
  - Expressões também possuem tipo
    - Em Java, se `a` é do tipo `double`, a expressão `a*0` tem o tipo `double` (o valor será `0.0`)

# Inferência de Tipos

- ML leva a inferência de tipos ao extremo, fazer a inferência estática de tipos de toda expressão e função (sem necessidade de anotações)
  - As expressões a seguir são equivalentes

```
fun area(l : real, w : real) : real = l * w;
```

```
fun area(l, w) : real = l * w;
```

```
fun area(l : real, w) = l * w;
```

  - Porém, a função a seguir não é equivalente às anteriores

```
fun area(l, w) = l * w;
```

Qual o tipo dessa última função?

# USOS PARA TIPOS > VERIFICAÇÃO DE TIPOS

# Verificação de Tipos

- Considere a expressão a seguir em ML

```
not x;
```

Qual o resultado da avaliação desta expressão?

  - Se o valor de `x` for do tipo `bool`, a operação pode ser feita

```
- val x = true;
val x = true : bool
- not x;
val it = false : bool
```

  - Caso contrário, se `x` for do tipo `string` por exemplo, a operação não pode ser feita

Quem faz esta verificação? E quando ela é feita?

```
- val x = "abc";
val x = "abc" : string
- not x;
stdIn:2.1-2.6 Error: operator and operand do
not agree [tycon mismatch]
```

# Verificação de Tipos

- Se todos os tipos são definidos estaticamente…
  - Então a verificação de tipos **pode ser** feita em **tempo de compilação**
  - **Verificação de tipo estática** (*static type checking*)

- Se os tipos são definidos dinamicamente...
  - Então a verificação de tipos **deve ser** feita em **tempo de execução**
  - **Verificação de tipo dinâmica** (*dynamic type checking*)

# Verificação de Tipo Estática

- Verificação de tipo estática determina o tipo de tudo antes de executar o programa (variáveis, funções, expressões, etc)
  - O sistema de tipos recebe informações de anotações ou inferências de tipos (normalmente de ambos)

- Quando os tipos estáticos não são consistentes, o sistema de linguagem gera uma mensagem de erro antes de rodar o programa
  - Operadores: `1 + "abc"`
  - Funções: `round("abc")`
  - Comandos: `if "abc" then ...`

A maioria das linguagens de programação são estaticamente tipadas

# Verificação de Tipo Dinâmica

- Verificação de tipo dinâmica é similar a verificação estática, mas feita durante a execução do programa
  - O sistema de tipos não consegue prever se haverão erros de tipos
- Antes de aplicar uma operação, o sistema de tipos verifica se os operandos são de tipos adequados para os operadores
  - Se não forem compatíveis, o sistema de tipos não realiza a operação e emite um erro em tempo de execução
- Para realizar a verificação dinamicamente, o sistema precisa reconhecer o tipo de um determinado valor
  - Ou seja, todos os valores são associados a um tipo em tempo de execução (o sistema precisa manter esta informação)

Quais as desvantagens de manter estas informações em tempo de execução?

# Verificação de Tipos Estática e Dinâmica

- Muitos sistemas de linguagens usam uma mistura de verificação estática e dinâmica de tipos
  - Subtipos em linguagens praticamente verificadas estaticamente precisam usar informações de tipos em tempo de execução
    - Ex.: o compilador pode saber estaticamente o tipo de um parâmetro formal, mas o valor realmente passado em uma chamada pode ser um subtipo (esta informação deve ser precisamente anotada durante a execução)
  - No outro lado do espectro, linguagens que são praticamente verificadas dinamicamente podem aproveitar de anotações e inferências de tipos feitas estaticamente
    - Ex.: Tipos estáticos podem ser inferidos em partes de Lisp; assim o compilador pode gerar códigos melhores eliminando verificações de tipos em tempo de execução



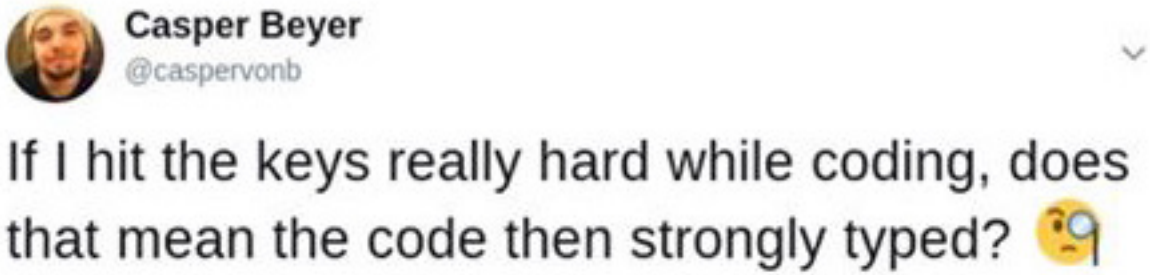
# Verificação de Tipos Estática e Dinâmica

- Linguagens que exigem a presença de informações de tipos em tempo de execução podem incluir construções para que programadores acessem estas informações
  - Em Java pode se testar o tipo de um objeto com `instanceof`
  - Em Modula-3 tem o comando `typecase` que desvia o código dependendo do tipo do objeto

# Linguagens Fortemente Tipadas

- O objetivo da verificação de tipos, seja estática ou dinâmica, é evitar a aplicação de operações com tipos errados de operandos

- Em algumas linguagens esta prevenção é forte o suficiente para se criar uma garantia
  - O sistema de linguagens faz verificações de tipos tão minuciosamente que nenhuma aplicação incorreta de tipos deixa de ser identificada





# Linguagens Fortemente Tipadas

- Linguagens que garantem esta propriedade são chamadas de linguagens fortemente tipadas
  - Java♣, ML e C# se enquadram nesta categoria

- Já outras linguagem possuem "buracos" no sistema de tipos que adicionam flexibilidade, mas enfraquecem esta garantia
  - FORTRAN 77: parâmetros, `EQUIVALENCE`
  - Pascal: registros variantes
  - C e C++: parâmetros, uniões
  - Ada: `UNCHECKED_CONVERSION`

♣: http://phrack.org/issues/70/7.html

# Coerção

- Regras de coerção têm um efeito importante na verificação de tipos

- Por exemplo, em Java, expressões são fortemente tipadas
  - Operar um inteiro (`int`) com ponto flutuante (`float`) é legal (o inteiro é convertido para ponto flutuante antes da operação)
  - Contudo, essa coerção resulta na perda de um dos benefícios da tipagem forte: a detecção de erro

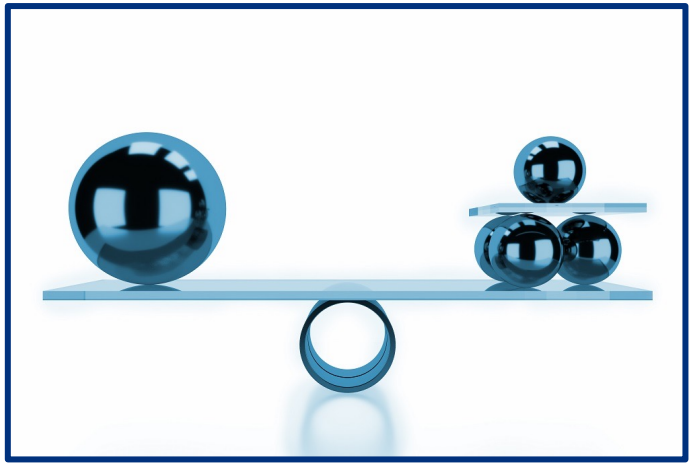
```
int a, b; // duas variáveis inteiras
float d;  // uma variável ponto flutuante

// se a intenção do programador era usar
// a + b, então o compilador não pegaria
// esse "erro de tipo"
... = a + d;
```

**Cuidado:** coerção enfraquece a tipagem forte

# Coerção

- Regras de coerção afetam consideravelmente linguagens fortemente tipadas enfraquecendo a verificação de tipos
- Linguagens com muitas coerções (como C ou C++) são menos confiáveis que linguagens com poucas coerções (como ADA) ou linguagens com nenhuma coerção (como ML e F#)
- Java e C# têm metade das coerções de C++, portanto sua detecção de erros é mais eficiente, mas não tão eficientes quanto as de ML e F#

# USOS PARA TIPOS > EQUIVALÊNCIA DE TIPOS



# Compatibilidade de Tipos

- A ideia de compatibilidade de tipos veio com a verificação de tipos
- As regras de compatibilidade ditam quais os tipos de operandos são aceitáveis para cada operador, portanto estabelecem os possíveis erros de tipos da linguagem
- Essas regras são chamadas de compatibilidade porque em alguns casos os operandos podem ser convertidos implicitamente pelo compilador ou pelo ambiente de execução para um operador

# Equivalência de Tipos

- As regras de compatibilidade são simples e rígidas para escalares, mas são complexas para tipos estruturados como arranjos e registros
  - Coerção desses tipos é rara, portanto o problema é mais de **equivalência de tipos** que de compatibilidade de tipos

- Dois tipos são **equivalentes** se um operando de um tipo em uma expressão pode ser substituído por outro tipo **sem coerção**

Equivalência de tipos é uma forma mais restrita de compatibilidade de tipos: compatibilidade sem coerção

Como definir equivalência de tipos?

# Equivalência de Tipos

- Duas variáveis são de tipos equivalentes se qualquer uma delas pode ter seu valor atribuído a outra

- Existem duas abordagens para definição de equivalência de tipo
  - **Equivalência de nome**
  - **Equivalência de estrutura**

# Equivalência de Nome

- Variáveis possuem tipos equivalentes se estiverem na mesma declaração ou em declarações que usem o mesmo nome de tipo
  - Vantagem: mais fácil de implementar
  - Desvantagens: altamente restritiva

- Exemplo
  - Compatível (em C)

```
int w = 10;
int c;

c = w;
```

  - Incompatível (em Pascal)

```
type indextype = 1..100;

var
   count : integer;
   index : indextype;

   count := index;
```

# Equivalência de Estrutura

- Variáveis têm tipos compatíveis se os seus tipos tiverem estruturas idênticas
  - Vantagens: mais flexibilidade
  - Desvantagens: mais difícil de implementar

- Exemplo (em C)

```
char letraA = 'A';
int ordemA = 65;

letraA = ordemA;
```

# Equivalência de Estrutura

- Comparar a estrutura dos tipos não é simples
  - Dois registros (`struct`) são equivalentes se os nomes de seus campos forem diferentes?
  - Dois arranjos unidimensionais, um com índice de `0..10` e outro de `1..11`, são equivalentes?
  - Duas enumerações com a mesma quantidade de componentes, mas com nomes diferentes, são equivalentes?

- Um outro problema é não permitir diferenciar tipos de uma mesma estrutura

```
type Celsius = Float;
     Fahrenheit = Float;
```

**Cuidado:** não deveriam ser misturados em expressões

# Abordagens em Linguagens

- Em C++ e Java
  - Utiliza equivalência de nomes
  - A compatibilidade de objetos está relacionada com a hierarquia de herança

- Em C
  - Utiliza equivalência de tipos por estrutura, enquanto `struct` e `union` por equivalência de nome...
  - ...mas se estiverem em arquivos diferentes é feita equivalência estrutural

`typedef` em C/C++ não cria um novo tipo, simplesmente dá um novo nome a um tipo existente

# Abordagens em Linguagens

- Em Pascal
  - Utiliza a equivalência de nomes

```
type
    type1 = array[1..50] of integer;
    type2 = array[1..50] of integer;
    type3 = type2;
```

  - Os tipos `type1` e `type2` são incompatíveis (já que pascal não possui equivalência de estrutura)
  - Já os tipos `type2` e `type3` são compatíveis (pela equivalência de nomes)

# ISSO É TUDO, PESSOAL!