# *Estruturas de Linguagem*

**Francisco Sant'Anna**

**francisco@ime.uerj.br**

**http://github.com/fsantanna/EDL**

# Tipos de Dados

- Coleção de valores
  - finitos: true, false
  - infinitos: [1], [1,2], [1,2,3], ...
- Operações
  - +, *not*, >, *contains*

# Tipos de Dados

- Primitivos
  - booleano, inteiro, float
- Compostos
  - listas, arrays, unions

# Tipos de Dados

- Definidos pela linguagem
- Definidos pelo programador
  - ```
    struct rect_t {
        int x, y, w, h;
    };
    ```
    ```

# Tipos de Dados

- Concretos
- Abstratos
  - Separação entre ***interface*** vs ***representação+operações*** (implementação)

```
// rect.c

struct rect_t {
  int x, y, w, h;
};

int getArea (struct rect_t* r) {
  return r->w * r->h;
}
```

```
// rect.h
struct rect_t;
int getArea (struct rect_t* r);
```

# Tipos de Dados

- Idealmente mapeiam para o problema/domínio
  - expressividade

- Média de notas da turma
  - `float media;`
- Notas da turma
  - `List<Tuple<string, Tuple<float,float>>>`

# Sistema de Tipos

of supporting a large range of applications. A better approach, introduced in ALGOL 68, is to provide a few basic types and a few flexible structure-defining operators that allow a programmer to design a data structure for each need. Clearly, this was one of the most important advances in the evolution of data type design. User-defined types also provide improved readability through the use of meaningful names for types. They allow type checking of the variables of a special category of use, which would otherwise not be possible. User-defined types also aid modifiability: A programmer can change the type of a category of variables in a program by changing a type definition statement only.

There are a number of uses of the type system of a programming language. The most practical of these is error detection. The process and value of type checking, which is directed by the type system of the language, are discussed in Section 6.12. A second use of a type system is the assistance it provides for program modularization. This results from the cross-module type checking that ensures the consistency of the interfaces among modules. Another use of a type system is documentation. The type declarations in a program document information about its data, which provides clues about the program's behavior.

# Sistema de Tipos

- Define a associação entre tipos e expressões
  - *binding time?*
- Inclui regras de equivalência

```
int v1 = 10;
void* ptr = &v1;
```

```
interface Shape {
   int getArea ();
}

class Rectangle implements Shape {
    ...
}

Shape s1 = new Rectangle();
int v = s1.getArea();
```

# Tipos Primitivos

- Reflexo do hardware
  - tipos ou operações?
- Numéricos
  - em diversos tamanhos
  - com ou sem sinal
  - inteiros grandes
    - 123456789012345678901
  - reais, float
  - complexos, decimais

exemplos

# Tipos Primitivos

- Caractere
  - ASCII, Unicode
- String
  - primitivos ou array de caracteres?
  - tamanho estático ou dinâmico?
  - constantes ou modificáveis?
  - concatenação, substring, pattern matching

# Tipos Compostos

# Arrays

- Conjunto de valores homogêneos (mesmo tipo)
- Elemento identificado pela posição (índice)
- Decisões de design
  - limites checados?
  - alocação estática, pilha, heap?
  - arrays multidimensionais?

# Arrays em Lua

- Caso particular do tipo `table`
- Criação

```
arr = {}
```

- Indexação

```
arr[i]
```

  - não assume limites

```
arr[-1]=1; print(arr[-1])
```

- Tamanho

```
#arr
```

  - somente para sequências começando em 1
  - array não esparsos
  - dinâmico

```
for i=1,1e9 do arr[i]=1 end
```

- Construtor

```
arr = { 5, 6, [3]=7 }
```