

# *Estruturas de Linguagem*

## *Abstração de Código Subprogramas*

**Francisco Sant'Anna**

**francisco@ime.uerj.br**

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

# O Conceito de Abstração

An **abstraction** is a view or representation of an entity that includes only the most significant attributes.

- Classes de abstrações

- processos

```
sortInt(list, listLen)
```

- dados

```
float d;
```

- Uma “arma” contra complexidade

- facilita no gerenciamento de programas

# Abstração de Código (definida pelo programador)

- “Subprogramas” definidos pelo programador
- Abstração para com sequência de instruções
- Detalhes desnecessários são escondidos
- O programa “chama” o subprograma, passando-o temporariamente o controle da CPU
- Economia de memória e tempo de desenvolvimento

# Características Gerais

- Um único ponto de entrada
- O chamador é suspenso enquanto o subprograma chamado executa
  - implica que só há um subprograma em execução
- O controle retorna ao chamador ao fim da execução
- Threads, Corrotinas, etc?

# Características Gerais

- Definição
  - nome
  - parâmetros de entrada
  - parâmetros de saída
- Implementação
  - sequência de instruções
- Chamada
  - nome
  - argumentos de entrada
  - argumentos de saída

```
function add (a,b)
    local ret = a + b
    return ret
end

local ret = add(10,20)
```

# Parâmetros

- Como “configurar” o subprograma?
  - acesso direto à não locais (globais, upvals, campos)
  - passagem de parâmetros
    - formal paramters vs actual parameters (arguments)
    - por posição ou por chave

```
function add (a,b)
  local ret = a + b
  return ret
end
```

```
ret = add(10,20)
ret = add(a=10,b=20)
```

```
function add (t)
  local ret = t.a + t.b
  return ret
end
```

```
ret = add({a=10,b=20})
ret = add{a=10,b=20}
```

# Parâmetros

- Passagem por posição ou por chave

```
function add (a,b)
  local ret = a + b
  return ret
end
```

```
ret = add(10,20)
ret = add(a=10,b=20)
```

```
function add (t)
  local ret = t.a + t.b
  return ret
end
```

```
ret = add({a=10,b=20})
ret = add{a=10,b=20}
```

# Parâmetros

- Valores padrão

```
float compute_pay(float income, float tax_rate,  
                  int exemptions = 1)
```

```
pay = compute_pay(20000.0, 0.15);
```

```
function add (a,b)  
    b = b or 20  
    local ret = a + b  
    return ret  
end
```

```
ret = add(10,20)  
ret = add(10)
```



# Procedimentos vs Funções

```
Stack* create (void) ;  
int      empty  (Stack*) ;  
void     push   (Stack*, int) ;  
void     pop    (Stack*) ;  
int      top    (Stack*) ;
```

```
create  : Stack  
empty   : Stack -> Bool  
push    : Int -> Stack -> Stack  
pop     : Stack -> Maybe Stack  
top     : Stack -> Maybe Int
```

# Considerações

- Are local variables statically or dynamically allocated?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter-passing method or methods are used?
- Are the types of the actual parameters checked against the types of the formal parameters?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Are functional side effects allowed?
- What types of values can be returned from functions?
- How many values can be returned from functions?
- Can subprograms be overloaded?
- Can subprograms be generic?
- If the language allows nested subprograms, are closures supported?

# Mecanismos de Passagem de Parâmetros

- Entrada (leitura), Saída (escrita), ou ambos?

```
int push_all (Stack* s1, Stack* s2);  
//out          //inout      //in
```

```
push_all (int ret, Stack* s1, Stack* s2);  
          //out      //inout      //in
```

- Passagem por Valor (*call-by-value*)
- Passagem por Resultado (*call-by-result*)
- Passagem por Referência (*call-by-reference*)
- Passagem por Nome (*call-by-name*)
- Passagem por Necessidade (*call-by-need*)

# Passagem por Valor

- Argumento passado é copiado para o parâmetro formal
- Somente modo de entrada
- Mais eficiente para tipos escalares

```
Stack s1, s2 = ...  
int ret;  
push_all(s1, s2, ret);
```

# Passagem por Resultado

- Resultado é copiado para o argumento
- Somente modo de saída
- Mais eficiente para tipos escalares
- Considerações

```
void Fixer(out int x,
    x = 17;
    y = 35;
}
. . .
f.Fixer(out a, out
```

```
void DoIt(out int x, int index) {
    x = 17;
    index = 42;
}
. . .
sub = 21;
f.DoIt(list[sub], sub);
```

# Passagem por Referência

- Identidade do argumento é passada para o parâmetro formal
- Modo de entrada e saída
- Indireção extra para todos os acessos
- Mais eficiente para tipos compostos

```
Stack s1,
```

```
int ret;
```

```
push_all(s
```

```
void fun(int &first, int &second)
```

If the call to fun happens to pass the same variable twice, as in

```
fun(total, total)
```

then first and second in fun will be aliases.

# Passagem por Nome

- O texto do argumento substitui o parâmetro formal
- Modo de entrada e saída
- Estilo macros de C

```
#define f(a,b) {  
    ret = a + (i++) + b;  
}  
f(v[i], v[i]);
```

# Passagem por Necessidade

- Argumento passado é copiado para o parâmetro formal
- Somente modo de entrada
- Similar à passagem por valor
- Adia a avaliação do argumento até o seu uso (*lazy evaluation*)

```
define g(a, b, c) = if a then b else c  
l = g(h, i, j)
```

```
1 : 1 : 1 ... and the expression let x = 1:x
```