

# *Estruturas de Controle*

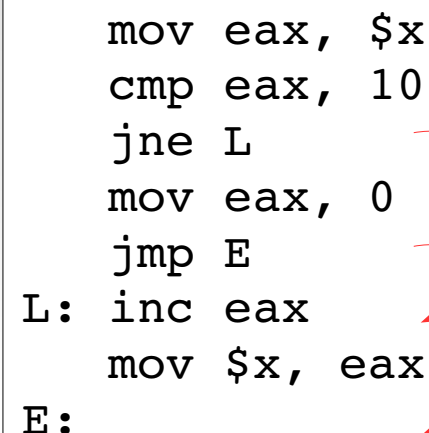
**Francisco Sant'Anna**

# Como entender o fluxo de execução de um programa a partir de seu código fonte?

- Linguagens de Máquina

- Estrutura plana de código
  - `<LABEL, INSTR, op1, ...>`
- Sequências e Saltos

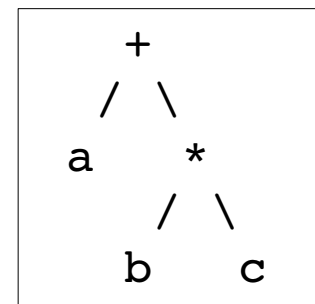
```
mov eax, $x
cmp eax, 10
jne L
mov eax, 0
jmp E
L: inc eax
   mov $x, eax
E:
```



- Linguagens de Alto Nível

- Estrutura hierárquica de código
  - Precedência de Operadores
  - Blocos de Controle
    - (indentação natural)
  - Rotinas (Subprogramas)

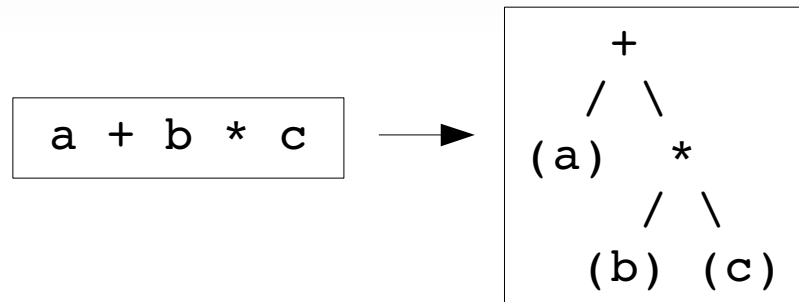
`a + b * c`



```
if (x == 10) then
    x = 0;           // (1)
else
    x = x + 1;      // (2)
end
```

# Expressões

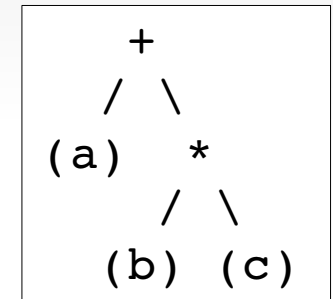
- Combinam valores através de operadores
  - Valor: “operando já avaliado”



# Operadores

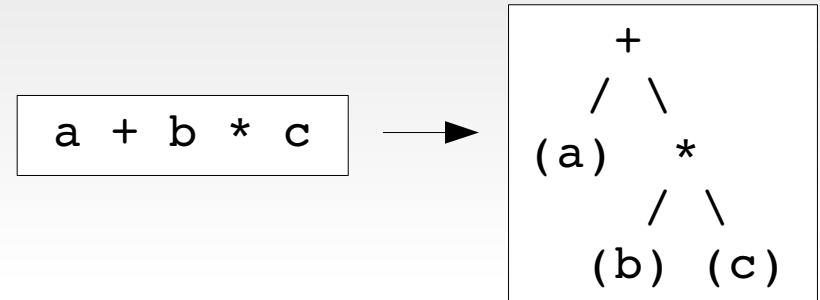
- Aridade (número de operandos):
  - Unário: `(not true) → false`      `(-10) → -10`
  - Binário: `(1+1) → 2`      `(12>10) → true`
  - N-ário: `max(10, 100, 1, 50) → 100`
- Notação:
  - infixada `[1+2]`
  - pré-fixada `[+ 1 2]`
  - pós-fixada `[1 2 +]`
- Precedência para notação infixada:
  - [unário, multiplicação, adição, ...]
  - **Cuidado!** `a==b<c` Pascal: `(a==b)<c` C: `a==(b<c)`
  - Uso de parênteses se sobrepõe à precedência

`a + b * c`



# Operandos

- Ordem de avaliação
  - Esquerda para a direita
  - Paralelo
  - Não especificado



- Cuidado! Em C a ordem não é especificada:

```
x = f (&v) + g (&v) ;
```

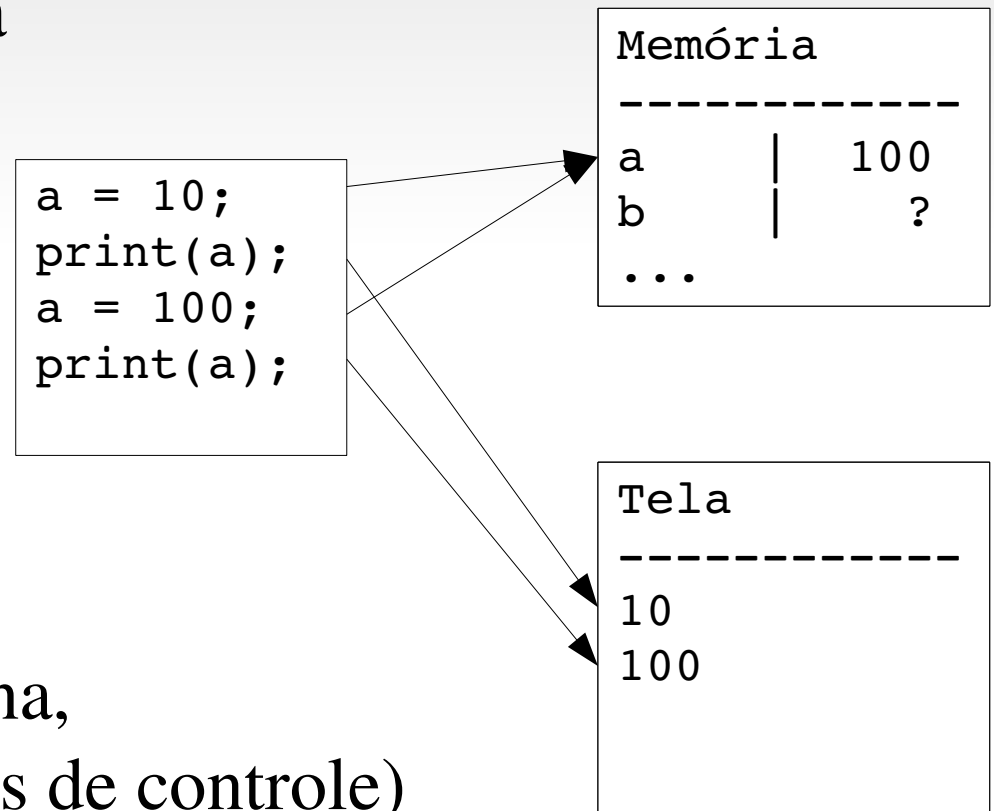
- a = f (&v) ;

```
b = g (&v) ;
```

```
x = a + b
```

# Comandos

- Executam em sequência
- Alteram o estado da máquina
  - *Efeito colateral*
  - Memória
    - `a = 10`
  - Dispositivo de saída
    - `print(a)`
- Atribuição, Chamada de rotina, Comandos compostos (blocos de controle)



# Condicionais

- Escolha entre diversos caminhos de execução possíveis
- Escolha se baseia no estado atual da máquina

- Memória

```
■ if (a == 10) then
    print('a é igual a 10')
else
    print('a é diferente de 10')
end
```

Memória		
-----		
a		100
b		?
...		

- Dispositivo de entrada

```
■ if (key() == 'x') then
    print('você pressionou x')
else
    print('você pressionou outra tecla')
end
```

Teclado	
-----	
x	

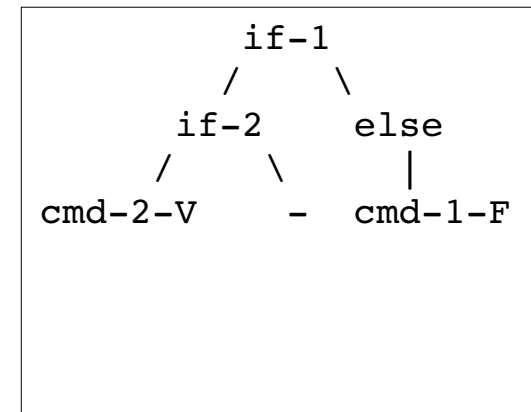
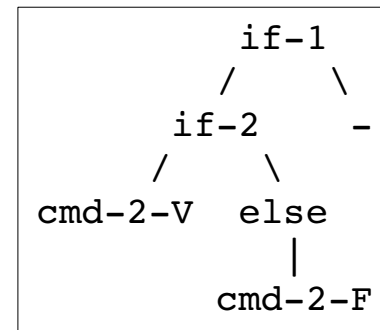
# Condicionais

- Escolha entre dois caminhos

- `if (<Exp-Condção>) {`  
    <Sequência-Verdadeiro>  
`}` `else {`  
    <Sequência-Falso>  
`}`

- Cuidado! Em C, uso sem delimitadores `{ }` é ambíguo:

- `if (<Exp-Condção-1>`  
    `if (<Exp-Condção-2>`  
        <Comando-2-Verdadeiro>;  
    `else`  
        <Comando- [1/2] - False>;





# Condicionais

- Escolha entre múltiplos caminhos

- ```
switch (<Exp-Condição>) {  
    case <Exp-1>:  
        <Sequência-V1>  
    <...>  
    case <Exp-N>:  
        <Sequência-VN>  
    default:  
        <Sequência-Alternativa>  
}
```



```
v = <Exp-Condição>  
if (v == <Exp-1>) then  
    <Sequência-V1>  
else  
    if (v == <Exp-2>) then  
        <Sequência-V2>  
    else  
        ...  
        else  
            <Sequência-Alt>  
        end  
        ...  
    end  
end
```

- Pode ser traduzido/entendido como if-else aninhados
- Porém, implementações otimizadas podem cortar caminhos

# Repetições (loops)

- Repetição de uma sequência de comandos
- Término se baseia em uma condição

```
a = 64;
```

```
while (a > 10) do
```

```
    print(a)
```

```
    a = a / 2
```

```
end
```

Memória

|             |   |    |
|-------------|---|----|
| <i>r1</i> : | a | 64 |
| <i>r2</i> : | a | 32 |
| <i>r3</i> : | a | 16 |
| <i>r4</i> : | a | 8  |

Tela

|             |    |
|-------------|----|
| <i>r1</i> : | 64 |
| <i>r2</i> : | 32 |
| <i>r3</i> : | 16 |

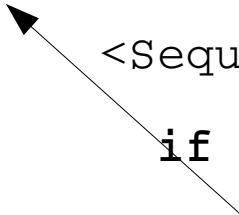
# Repetições (loops)

- Repetição infinita

```
while (true) do  
    <Sequência-Repetida-Para-Sempre>  
end
```

- Quebra/escape de repetição

```
while (<Exp>) do  
    <Sequência-Repetida>  
    if <Exp-2> then  
        break; // força a saída do loop "mais próximo"  
    end  
end
```

A diagram consisting of a black arrow pointing from the `break;` statement inside the `if` block to the `while` loop header, indicating that the loop is terminated.


# Repetições (loops)

- Iteradores: variável de controle sobre um conjunto de valores
- Numéricos: ex., de 1 até 5

```
v = 0
for i=1, 5 do
    v = v + i; // v=1+2+3+4+5
end
print(v)      // 15
```

- Genéricos (geradores)

```
it.start()
while (it.hasMore()) do
    v = it.next()
    <use-v>
end
```



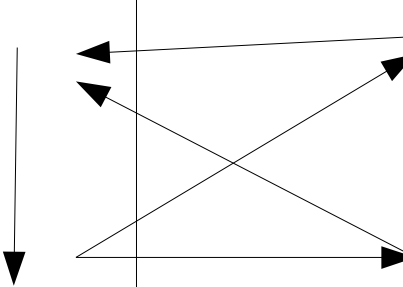
```
for linha in key.lines() do
    print(linha)
end
```

# Rotinas

- Mecanismo de decomposição de programas
  1. Associa um nome a uma sequência de comandos
  2. Chama esse nome de outras partes do programa
- Exemplo:
  1. Associa o nome “imprimeSoma” à sequência que imprime a soma de 1 a 5
  2. Chama “imprimeSoma()” de outras partes do código

```
def imprimeSoma do
  v = 0
  for i=1, 5 do
    v = v + i;
  end
  print(v)
end
```

```
imprimeSoma() // 15
...
imprimeSoma() // 15
```

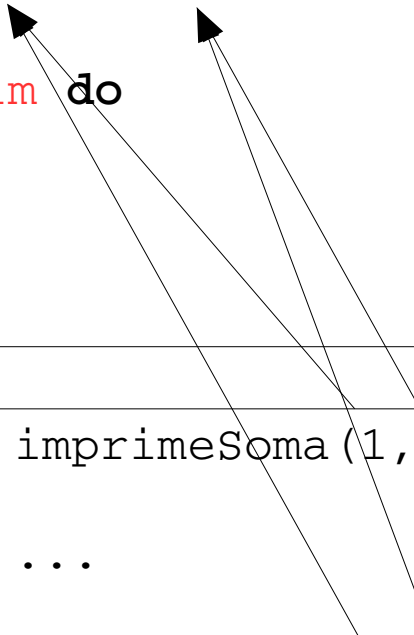


# Rotinas

- Parâmetros
  - Permitem customizar o comportamento da rotina

```
def imprimeSoma (inicio, fim) do
  v = 0
  for i=inicio, fim do
    v = v + i;
  end
  print(v)
end
```

```
imprimeSoma(1,5)    // 15
...
imprimeSoma(4,5)    // 9
```



The diagram consists of two arrows pointing from the first parameter '1' in the first function call to the parameter 'inicio' in the function definition. Another two arrows point from the second parameter '5' in the first function call to the parameter 'fim' in the function definition. A similar pair of arrows connects the '4' and '5' in the second function call to the 'inicio' and 'fim' parameters in the definition, respectively.

# Rotinas

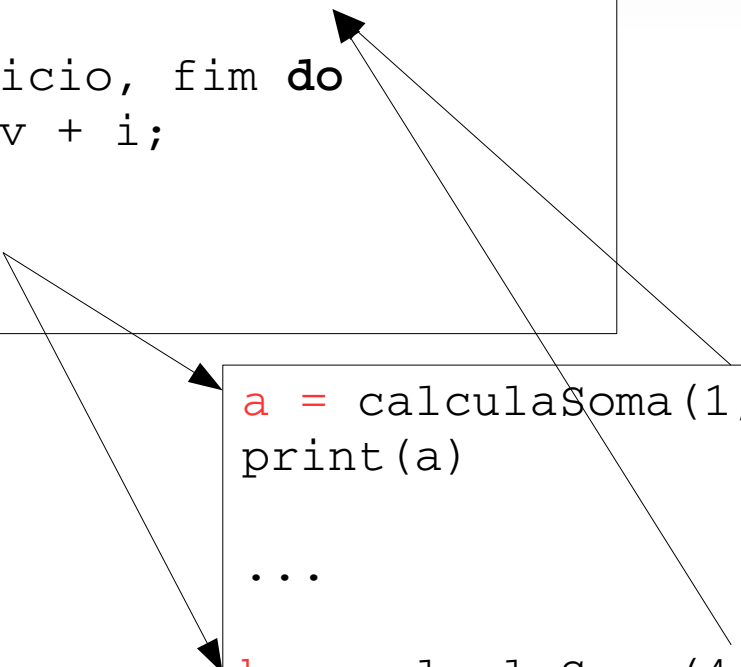
- Retorno
  - Permite que a rotina retorne um resultado a quem a chamou

```
def calculaSoma (inicio, fim) do
  v = 0
  for i=inicio, fim do
    v = v + i;
  end
  return v
end
```

```
a = calculaSoma(1,5)
print(a)                // 15

...

b = calculaSoma(4,5)
send(b)                 // 9
```



# Rotinas

- Parâmetros
  - Passagem por valor e por referência
- Exemplo: rotina para fazer duas variáveis trocarem de valor entre si

```
def trocaVal (val a, b) do
  tmp = a
  a    = b
  b    = tmp
end
```

```
def trocaRef (ref a, b) do
  tmp = a
  a    = b
  b    = tmp
end
```

```
x = 1
y = 2
trocaVal(x,y)
print(x,y) // 1,2
trocaRef(x,y)
print(x,y) // 2,1
```

Valores de x,y  
(r-value)

Endereço de x,y  
(l-value)

| Memória |  |   |
|---------|--|---|
| -----   |  |   |
| x       |  | 1 |
| y       |  | 2 |
| -----   |  |   |
| a       |  | 2 |
| b       |  | 1 |

| Memória |  |    |
|---------|--|----|
| -----   |  |    |
| x       |  | 2  |
| y       |  | 1  |
| -----   |  |    |
| a       |  | &x |
| b       |  | &y |



# Rotinas

- Procedimentos e Funções

- Procedimentos

- produzem efeitos globais
    - parâmetros por valor e referência
    - sem retorno

- Funções

- não produzem efeitos globais
    - parâmetros apenas por valor
    - com retorno

```
// procedimento
def imprimeSoma (ini, fim) do
  v = 0
  for i=ini, fim do
    v = v + i;
  end
  print(v)
end
```

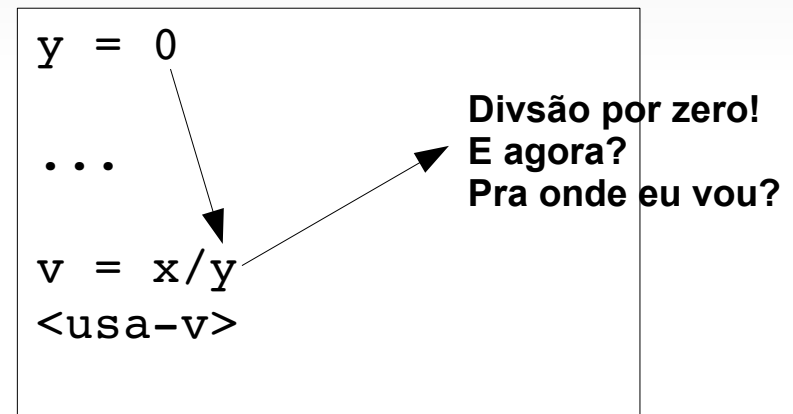
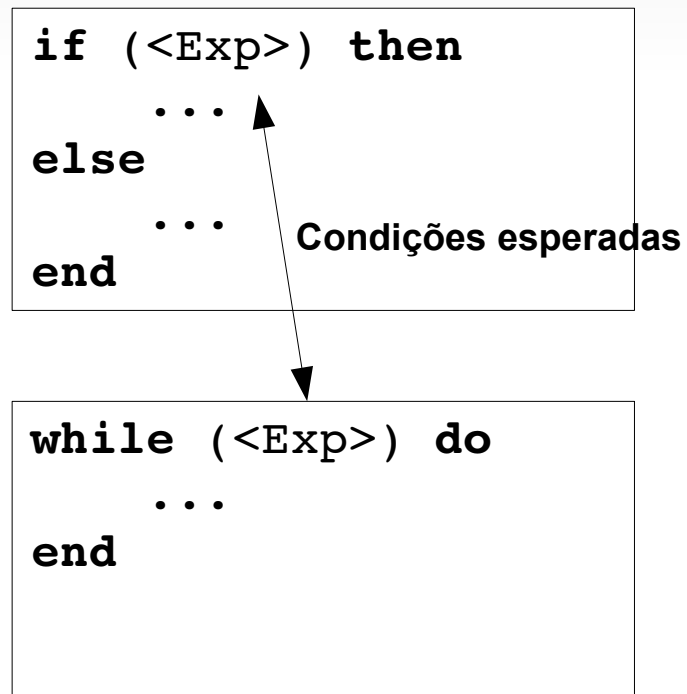
```
// função
def calculaSoma (ini, fim) do
  v = 0
  for i=ini, fim do
    v = v + i;
  end
  return v
end
```

# Rotinas

- Procedimentos + Funções
  - Conveniência: `print(v)`
  - Flexibilidade: `swap(x,y)`
  - Entendimento: `f(x) + x`
  - Otimizações: `f(x) + f(x)`
    - “*aliasing*”

# Exceções

- Condições não esperadas
  - Quebram o fluxo normal de execução



# Exceções

- Exceções típicas
  - Divisão por zero, raiz de número negativo
  - Acesso a vetor for a dos limites
  - Falta de memória
- Efeitos diversos
  - Erro de execução (acessos for de limite, em Pascal)
  - Não especificado (divisão por 0, em C)
  - **Tratamento apropriado** (*exceções têm tratamento primitivo*)
    - Confiabilidade
    - **Java, C++, Eiffel**

# Exceções

1. Quais exceções podem ser tratadas?
2. Quem pode gerar uma exceção e como?
3. Quem pode tratar uma exceção?
4. Como é o controle de fluxo antes, durante e depois do tratamento?

- Quais exceções podem ser tratadas?
  - Pré-definidas: divisão por zero, falta de memória, etc
  - Definidas explicitamente pelo programador
    - Com valores associados?
  - Definidas implicitamente pelo programador/linguagem
    - Contratos: pré-condições, pós-condições e invariantes

```
// Java
class ExcecaoPilha extends Exception {
    public String msg;
    public ExcecaoPilha(String msg) {
        this.msg = msg;
    }
}
```

```
// Eiffel
is_empty: Boolean
    <verdadeiro se pilha vazia>
push (i: Item)
    <empilha i>
    ensure: not is_empty
pop: Item
    <remove e retorna o topo>
    require: not is_empty
```

- Quem pode gerar uma exceção e como?
  - *Runtime* da linguagem, implicitamente
    - Java: divisão por zero, falta de memória, etc
    - Eiffel: falha de contrato
  - Programador, explicitamente
    - Java: **throw** dentro de métodos anotados com **throws**

```
// Java
public int pop () throws ExcecaoPilha {
    if (this.isEmpty()) {
        throw new ExcecaoPilha("pop");
    }
    ...
}
public void push (int v) throws ExcecaoPilha {
    ...
    if (this.isEmpty()) {
        throw new ExcecaoPilha("push");
    }
}
```

```
// Eiffel (pilha)
is_empty: Boolean
    <verdadeiro se pilha vazia>
push (i: Item)
    <empilha i>
    ensure: not is_empty
pop: Item
    <remove e retorna o topo>
    require: not is_empty
```

- Quem pode tratar uma exceção e como?
  - Explicitamente pelo programador
    - Java: **try-catch**
    - Eiffel: **rescue**

```
// Java
public void f () {
    try {
        pop();
    } catch () {
        push(10);
    }
}
```

```
// Eiffel
do
    pop();
rescue
    push(10);
end
```



- Como é o controle de fluxo antes, durante e depois do tratamento
  - Java: throw → pilha desfeita → catch
  - Eiffel: falha de contrato → pilha desfeita → rescue/retry

```
f()                // trata a exceção (catch)
  g()              // não trata
    h()            // não trata
      throw();     // gera a exceção
    -
  -
catch()
```

```
// Java
public void f () {
  while (1) {
    try {
      pop();
      break();
    } catch () {
      push(10);
    }
  }
}
```

```
// Eiffel
do
  pop();
rescue
  push(10);
retry;
end
```

# Estruturas de Controle

- Fluxo de execução dentro do programa
- Estruturas Básicas de Controle
  - Expressões e Comandos
  - Condicionais e Iterações
- Chamada de rotinas
  - Recursões, chamadas de calda
- Tratamento de Exceções

# *Estruturas de Controle*

**Francisco Sant'Anna**