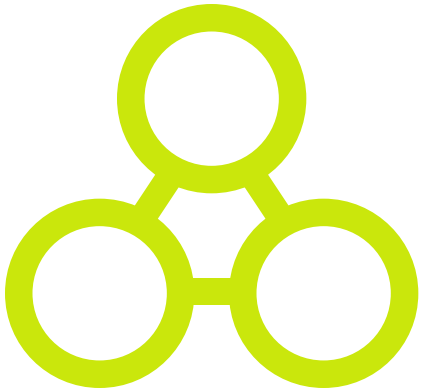


nemo

ontology & conceptual modeling research group

Linguagens de Programação

5 – Expressões e Comandos

Vítor E. Silva Souza

(vitor.souza@ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

Conteúdo do curso

- Introdução;
- Amarrações;
- Valores e tipos de dados;
- Variáveis e constantes;
- Expressões e comandos;
- Modularização;
- Polimorfismo;
- Exceções;
- Concorrência;
- Avaliação de linguagens.

- Estes slides foram baseados em:
  - Slides do prof. Flávio M. Varejão;
  - Livro “Linguagens de Programação – Conceitos e Técnicas” (Varejão);
  - Livro “Linguagens de Programação – Princípios e Paradigmas, 2a edição” (Tucker & Noonan).

Expressões

- Uma expressão é uma frase do programa que necessita ser avaliada e produz como resultado um valor:
  - Uma “conta”, uma “computação”;
- Elementos:
  - Operadores;
  - Operandos;
  - Resultado;
- Podem ser classificadas em:
  - Expressões simples: apenas um operador;
  - Expressões compostas: mais de um operador.

Operadores

- Quanto à **aridade**:
  - Unários (1 operando), binários (2), ternários (3), etc.;
  - Eneários: aridade variável (varargs C / C++ / Java);
  - Aridade elevada reduz legibilidade.
- Quanto à **origem**:
  - Pré-existentes (normalmente unários e binários);
  - Definidos pelo programador (funções, qualquer aridade);
  - Composição de operadores (ML e APL):

```
val par = fn (n: int) => (n mod 2 = 0)
val negacao = fn (t: bool) => if t then false else true
val impar = negacao o par
```

Operadores

- Quanto à **notação**:
  - Prefixada: operador antes dos operandos:

```
! a; --b; -10
```

  - Infixada / interfixada: operador entre os operandos;

```
a + b; c * d; 15 - 5
```

  - Posfixada: operador após os operandos.

```
a++; b--;
```

- Dialetos de Lisp permitem o uso de notação infixada ou prefixada (Notação de Cambridge) para operadores aritméticos: `+ a b`.

Tipos de expressão

- Podemos classificar expressões em diferentes tipos:
  - Literais, agregações, aritméticas, relacionais, booleanas, binárias, condicionais, etc.
- O tipo mais simples de expressão são os **literais**;
- Exemplos em C:

```
99.0      99      0143      'c'      0x43
```

Agregações

- Subtipo de expressão composta;
- Constrói um valor a partir de seus componentes:

```
int c[] = {1, 2, 3};
struct data d = {1, 7, 1999};
char *x = {'a', 'b', 'c', '\0'};
int b[6] = {0};
char *y = "abc";
```

- Agregações podem ser estáticas ou dinâmicas:

```
void f(int i) {
  int a[] = {3 + 5, 2, 16/4};          // Estática
  int b[] = {3 * i, 4 * i, 5 * i};     // Dinâmica
  int c[] = {i + 2, 3 + 4, 2 * i};     // Mista
}
```

Agregações

- Em C, a agregação só pode ser feita em operações de inicialização e no caso de strings constantes;
- Outras linguagens (ex.: Ada) são mais flexíveis:

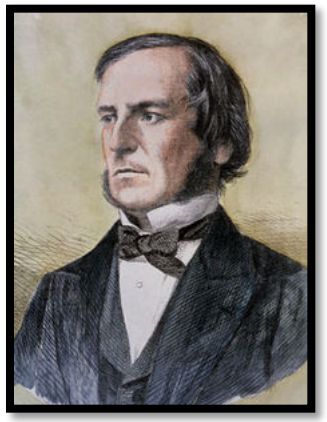
```
type data is record
    dia : integer range 1..31;
    mes : integer range 1..12;
    ano : integer range 1900..2100;
end record;

aniversario: data;
data_admissao: data := (29, 9, 1989);
aniversario := (28, 1, 2001);
data_admissao  := (dia => 5, ano => 1980, mes => 2);
```

Aritméticas, relacionais e booleanas

- Aritméticas:

```
float f;
int num = -9;   // - unário.
f = num / 6;    // Divisão inteira.
f = num / 6.0;  // Divisão de ponto flutuante.
```

- Relacionais:
  - Usadas para comparar os valores dos operandos;
- Booleanas:
  - Realizam as operações de negação, conjunção e disjunção da álgebra de Boole.



George Boole

Binárias

- Operações lógicas bit a bit:

```
int main() {
     int j = 10;
     char c = 2;
     printf("%d\n", ~0);       /* imprime -1 */
     printf("%d\n", j & c);    /* imprime  2 */
     printf("%d\n", j | c);    /* imprime 10 */
     printf("%d\n", j ^ c);    /* imprime  8 */
     printf("%d\n", j << c);   /* imprime 40 */
     printf("%d\n", j >> c);   /* imprime  2 */
  }
```

Alguém conhece o operador >>> de Java?

Binárias

- Operador >>> de Java:

```
public class Teste {
  private static void print(int val) {
     /* Imprime o número em formato binário formatado
         e também em formato decimal. */
  }

  public static void main(String[] args) {
     int i = -10;     print(i);
     i = i >> 1;      print(i);
     i = -10 >>> 1;   print(i);
  }
}
```

```
1111_1111_1111_1111_1111_1111_1111_0110 == -10
1111_1111_1111_1111_1111_1111_1111_1011 == -5
0111_1111_1111_1111_1111_1111_1111_1011 == 2147483643
```

Condicionais

- Exemplo em ML:

```
val c = if a > b  then a - 3 else b + 5
val k = case i of
      1 => if j > 5 then j - 8 else j + 5
    | 2 => 2*j
    | 3 => 3*j
    | _ => j
```

- Exemplo em C / C++ / Java:

```
max = x > y ? x : y;
```

- Algumas LPs (ex.: Ada) não oferecem expressões condicionais, forçando o uso de comandos (if):

```
if x > y then max := x; else max := y; end if;
```

Chamadas de função

- Também são consideradas expressões:
  - Operador = nome da função;
  - Operandos = parâmetros;
  - Resultado = retorno da função;
- Chamada condicional de função em ML:

```
val taxa =
  (if difPgVen > 0 then desconto else multa)(difPgVen)
```

- Simulando essa mesma construção em C:

```
// Ponteiro pra função:
double (*p)(double);
p = difPgVen < 0 ? desconto : multa;
taxa = (*p)(difPgVen);
```

Funções como cidadãs de 1ª classe

- Closures em Groovy e Java:

```
def obterClosure() {
  def valor = 10;
  return { println valor * 2 };
}
def closure = obterClosure();
closure();
```

```
import java.util.function.*;
public class Teste {
  public static Boolean positivo(Integer n) {
     return (n != null) && (n > 0);
  }
  public static void main(String ... args) {
     Function<Integer, Boolean> fun = Teste::positivo;
     System.out.println(fun.apply(-4));
  }
}
```

Operadores vs. funções

- Operadores das LPs denotam funções:

Expressão	Representação prefixada
`a * b`	`* (a, b)`
`c / d`	`/ (c, d)`
`a * b + c / d`	`+ (* (a, b), / (c, d))`

- Operadores possuem assinaturas (exemplos Java):

Operador	Assinatura da função
!	`[boolean -> boolean]`
&&	`[boolean x boolean -> boolean]`
*	`[int x int -> int],` `[float x float -> float], etc.`

- Ortogonalidade: Java não permite sobrescrita de operadores (C++, Ada, ML permitem).

Expressões com efeitos colaterais

- O objetivo principal de uma expressão é retornar valor;
- No entanto, algumas possuem outros efeitos:

```
// Em C: ++c retorna o valor de c+1,
// mas também incrementa c.
x = 3.2 * ++c;
```

- Podem gerar indeterminismo (depende do compilador):

```
// O que imprime o código C abaixo?
int x = 2;
int y = 4;
int z = (y = 2 * x + 1) + y;
printf("%d, %d, %d\n", x, y, z);
```

2, 5, 10

O compilador tenta avisar do perigo:
warning: unsequenced modification and access to 'y' [-Wunsequenced]

Expressões com efeitos colaterais

- Outro exemplo em C. Qual o valor de `i` ao final?

```
int i = 50;
for (int j = 0; j < 50; j++) i = i++;
printf("%d\n", i);
```

50

- Funções possibilitam a ocorrência de efeitos colaterais:

```
// Returns the character currently pointed by the
// internal file position indicator of the specified
// stream. The internal file position indicator is then
// advanced to the next character.
fgetc(f);
```

Em C, funções com tipo de retorno `void` são funções cujo único objetivo é produzir efeitos colaterais ou são considerados comandos?

Referenciamento

- Expressões usadas para acessar o conteúdo ou retornar referência para variáveis ou constantes:

```
*q = *q + 3;
const float pi = 3.1416;
int raio = 3;
float perimetro = 2 * pi * raio;
p[i] = p[i + 1];
*q = *q + 3;
r.ano = r.ano + 1;
s->dia = s->dia +1;
t = &m;
```

Referenciamento direto

Referenciamento indireto (a.k.a. derreferenciamento)

Operador **new** de Java/C++ também é de referenciamento, porém com efeito colateral. Qual seria esse efeito?

Referenciamento

Operador	Significado
[]	Acesso a valor ou retorno de referência de elemento de vetor.
*	Acesso a valor ou retorno de referência de variável ou constante apontada por ponteiro.
.	Acesso a valor ou retorno de referência de elemento de estrutura.
->	Acesso a valor ou retorno de referência de elemento de estrutura apontada por ponteiro.
&	Retorno de referência a qualquer tipo de variável ou constante.

Categóricas

- Realizam operações sobre tipos de dados:
  - Tamanho do tipo:

```
int i;
float *p = (float *)malloc(10 * sizeof (float));
int c[] = {1, 2, 3, 4, 5};

// Tam. do vetor (20) / tam. elem. (4) = num. elementos.
for (i = 0; i < sizeof c / sizeof *c; i++) c[i]++;
```

  - Conversão de tipo:

```
float f;
int num = 9, den = 5;
f = (float)num/den;
```

Categóricas

- Realizam operações sobre tipos de dados:
  - Identificação de tipo:

```
Profissao p = new Engenheiro();

if (p instanceof Medico)
  System.out.println("Registre-se no CRM");

if (p instanceof Engenheiro)
  System.out.println("Registre-se no CREA");
```

Avaliação de expressões compostas

- Precedência de Operadores
  - Escolha inadequada pode afetar a redigibilidade:

```
(* Erro em Pascal: and tem precedência sobre > e < *)
if a > 5 and b < 10 then

(* OK. Precedência ajustada com parênteses. *)
if (a > 5) and (b < 10) then
```

  - Porém, memorizar a ordem dificulta o aprendizado;
  - Ausência de precedência (Smalltalk e APL) baixa a redigibilidade;
  - Parênteses asseguram a ordem, mas reduzem redigibilidade e impedem otimizações.

Avaliação de expressões compostas

- Associatividade de operadores:
  - Operadores de mesma precedência;
  - Normalmente da esquerda para a direita:

```
x = a + b - c;
y = a < b < c;  // Compila em C, não em Java. Por que?
```

  - Podem existir exceções a essa regra:

```
x = **p;            // *(*p)
if (!!x) y = 3;     // !(!x)
a = b = c;          // a = (b = c);
```

  - APL não tem precedência e sempre associa da direita para a esquerda:

```
X = Y ÷ W - Z
```

Avaliação de expressões compostas

- Associatividade de operadores:
  - Em Fortran, exponenciação é da direita pra esquerda. Intuitivo ou não? Programadores vs. matemáticos;
  - Compiladores podem otimizar, mas isso pode causar problemas:

```
// Suponha f() e h() retornam números positivos grandes
// enquanto g() retorna um número negativo grande.
x = f() + g() + h();
```

Avaliação de expressões compostas

- Precedência de operandos:
  - Não determinismo em expressões

```
a[i] = i++;

int i = 50;
for (int j = 0; j < 50; j++) i = i++;
printf("%d\n", i); // 50 (!)
```

- Java resolve adotando precedência de operandos da esquerda para direita;
- Garante portabilidade, mas compromete eficiência (impede otimizações específicas de plataforma).

Avaliação de expressões compostas

- Expressões com curto-circuito:
  - Situação potencial (mas não vale a pena):

```
// Se x == y, o resultado é zero (nem precisa calcular).
z = (x - y) * (a + b) * (c - d);
```

  - Geralmente é usado em expressões booleanas:

```
int[] a = new int[n];
i = 0;
while (i < n && a[i] != v) i++;
```

  - Java e Ada tem operadores específicos para avaliação com (&&, ||) e sem (&, |) curto circuito;
  - Pascal não possui curto-circuito. Alguns compiladores permitem ativá-lo, mas então é usado sempre.

Avaliação de expressões compostas

- Expressões com curto-circuito:
  - Operadores booleanos de C e C++ usam curto circuito;
  - Pode-se usar operadores binários & e | pois não há curto circuito;
  - Curto circuito com efeitos colaterais reduz legibilidade:

```
if (b < 2 * c || a[i++] > c ) {
  a[i]++;
}
```

Como fica a propriedade de comutatividade das expressões lógicas?

Comandos

- Objetivo é atualizar variáveis ou controlar o fluxo de controle;
- Característicos de LPs imperativas;
- Podem ser primitivos ou compostos;
- Bastam atribuição, seleção e desvio, porém LP fica pouco expressiva.

Tipos de comando

- De acordo com [Watt, 1990]:
  - Atribuições;
  - Comandos sequenciais;
  - Comandos colaterais;
  - Comandos condicionais;
  - Comandos iterativos;
  - Chamadas de procedimento;
  - Comandos de desvio incondicional.

Atribuição

- = vs. :=
  - Ada, Pascal, Modula2 e APL (exemplo abaixo) usam := para não confundir com = (matemática):

```
i := !i + 1
if (a = 10) a += 3;
```

Derreferenciamento explícito: i amarrado a 2 conceitos diferentes.

- Atribuição simples:

```
a = b + 3 * c;
```

- Atribuição múltipla:

```
a = b = 0;
```

Neste caso, expressão ou comando?

- Atribuição condicional (ML):

```
(if a < b then a else b) := 2;
```

Atribuição

- Atribuição composta:

```
a += 3;
a *= 3;
a &= 3;
```

- Atribuição unária:

```
++a;
a++;
--a;
a--;
```

- Atribuição como expressão:

```
// + redigibilidade, - legibilidade
while ((ch = getchar()) != EOF) {
  printf("%c", ch);
}
```

Sequenciais

- Considerar um conjunto de comandos como um só:

```
if (x > y) printf("x > y\n");    // Um comando.

if (x > y) {                     // Bloco de comandos.
  n = 1;
  n += 3;
  if (n < 5) {
     n = 10;
     m = n * 2;
  }
}
```

- Delimitadores de bloco: { / }, begin / end, indentação, etc.

Colaterais

- Comandos que permitem processamento paralelo;
- Muito raros em LPs (exceção em LP recente: Go);
- Exemplo em ML:

```
a = 0;
a = 3, a = a + 1;

val altura = 2
and largura = 3
and comprimento = 5

and volume = altura * largura * comprimento
```

  - A última parte gera erro, pois usa identificadores presentes no mesmo comando colateral, o que é vetado pela linguagem.

Condicionais

- Seleção de caminho condicionado:

```
if (x < 0) { x = y + 2; x++; }
```

- Seleção de caminho duplo:

```
if (x < 0) { x = y + 2; x++; } else { x = y; x--; }
```

- Potencial problema com marcadores:

```
if ( x == 7 )
  if ( y == 11) {
     z = 13;
     w = 2;
  }
else z = 17;
```

```
if ( x == 7 ) {
  if ( y == 11) {
     z = 13;
     w = 2;
  }
} else z = 17;
```

```
if x > 0 then
  if y > 0 then
     z := 0;
  end if;
else
  z := 1;
end if;
```

Em Ada os marcadores são obrigatórios.

Condicionais

- Seleção de caminhos múltiplos:

```
switch (nota) {
  case 10:
  case 9: printf ("Muito Bom!!!");
          break;
  case 8:
  case 7: printf ("Bom!");
          break;
  case 6:
  case 5: printf ("Passou...");
          break;
  default: printf ("Estudar mais!");
}
```

  - Fortran: goto;
  - Ada e Pascal: switch sem break;
  - Python: não possui (purismo OO).

Condicionais

- Caminhos múltiplos com ifs aninhados:

```
if (rendaMes < 1000)
  iR = 0;
else if (rendaMes < 2000)
  iR = 0.15 * (2000 - rendaMes);
else
  iR = 0.275 * (rendaMes - 2000) +
    0.15 * (2000 – rendaMes);
```

  - Modula2, Ada e Fortran-90 têm `elsif`.

Iterativos

- Número indefinido de repetições:
  - Pré-teste e pós-teste:

```
f = 1;
y = x;
while (y > 0) {
  f = f * y;
  y--;
}
```

```
f = 1;
y = 1;
do {
  f = f * y;
  y++;
} while (y <= x);
```

Iterativos

- Pré-teste vs. pós-teste:

```
s = 0;
printf ("n: ");
scanf ("%d", &n);
while (n > 0) {
  s +=n;
  printf ("n: ");
  scanf ("%d", &n);
}
```

Repetição da leitura.

```
s = 0;
do {
  printf ("n: ");
  scanf ("%d", &n);
  if (n > 0) s+=n;
} while (n > 0);
```

Repetição da verificação n > 0.

Iterativos

- Número definido de repetições:
  - Em Modula-2:

```
s := 0;
FOR i := 10 * j TO 10 * (j + 1) BY j DO
  s := s + i;
END;
```

- Consenso:
  - Valores da variável de controle conhecidos antes do primeiro ciclo e fixos;
  - Realização do teste antes da execução do corpo.

Iterativos

- Variação no escopo da variável de controle:
  - Ada e Java restringem ao corpo;
  - Ada não permite alteração da variável de controle no corpo da repetição;
  - Fortran, Pascal e C tratam como variável ordinária;
  - C++ permite que escopo comece no comando.

Algumas LPs não especificam se a variável de controle pode ser usada como variável comum após a repetição, deixando a cargo do implementador do compilador (-portabilidade).

Iterativos

- Número definido de repetições:
  - Em C:

```
dif = 0;
for (i = 0; i < n; i++) {
  if (a[i] % 2 == 0) dif += a[i];
  if (a[i] % 3 == 0) dif -= a[i];
}

for (i = 10 * j, s = 0; i <= 10 * (j + 1); s += i++);

for (i = 0, s = 0; i <= n && s < 101 && a[i] > 0 ; )
  s += a[i++];

for (;;);
```

Iterativos

- Número definido de repetições:
  - Pode não se restringir a tipos primitivos discretos:

```
@dias = ("Dom", "Seg", "Ter", "Qua","Qui", "Sex", Sab");
foreach $dia (@dias) {
  print $dia
}
```

  - Maioria das LPs não oferece;
  - Java e C++ oferecem iteradores associados a coleções;
  - Java (>= 5) e C++ oferecem for-each.

Chamadas de procedimentos

- Objetivo é atualizar variáveis;
- Aplica-se aos procedimentos a mesma discussão feita com funções anteriormente (ex.: aridade);
- Em algumas LPs, substituído por função que retorna um valor vazio (`void`).

Desvios incondicionais

- Somente comandos de entrada e saída única podem ser restritivos em algumas situações;
- Entrada única e saídas múltiplas é positivo;
- Entradas múltiplas é negativo;
- Tipos:
  - Desvios irrestritos;
  - Escapes.

Desvio irrestrito

- Conhecido como comando goto;
- Pode ser nocivo à boa programação;
- Algumas LPs o eliminaram (Modula2, Java);
  - Em Java, goto é palavra reservada;
  - Combinação de escape (rotulado) e exceções tem sido suficiente para não precisar de goto;
- É importante em algumas situações;
- Também é usado para propagação de erros em LPs sem tratamento de exceções:
  - Visual Basic: On Error GoTo X

Necessidade do goto

```
achou = 0;
for (i = 0; i < n && !achou; i++)
  for (j = 0; j < n && !achou; j++)
    if (a[i] == b[j]) achou = 1;
if (achou) printf("achou!!!");
else printf("não achou!!!");
```

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    if (a[i] == b[j]) goto saida;

saida:
if (i < n) printf("achou!!!");
else printf("não achou!!!");
```

Escapes

- Desvios incondicionais estruturados;
- Não podem criar ou entrar em repetições:

```
s = 0;
for(;;) {
  printf ("n: ");
  scanf ("%d", &n);
  if (n <= 0) break;
  s+=n;
}
```

```
i = 0;
s = 0;
while(i < 10) {
  printf ("n: ");
  scanf ("%d", &n);
  if (n < 0) continue;
  s+=n;
  i++;
}
```

Escapes

- Associação com iterações rotuladas pode ser útil;
- Ada e Java oferecem:

```
saida:
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
     if (a[i] < b[j]) continue saida;
     if (a[i] == b[j]) break saida;
  }
}
if (i < n) printf ("achou!!!");
else printf ("não achou!!!");
```

Escapes

- Podem interromper a execução de subprogramas e programas:

```
void trata (int erro) {
  if (erro == 0) {
     printf (“nada a tratar!!!”);
     return;
  }

  if (erro < 0) {
     printf (“erro grave – nada a fazer!!!”);
     exit (1);   // O SO recebe o código de erro.
  }

  printf(“erro tratado!!!”);
}
```

Conclusões

- LP fica empobrecida quando não oferece tipos de expressões ou comandos discutidos nesta aula;
  - Porém, se é Turing-complete, é Turing-complete...
  - Ex.: Java com *closures* e sem *closures*;
- Expressões e comandos adicionais podem não acrescentar nada:
  - Entrada e saída em COBOL e Fortran vs. chamadas de procedimento;
- Existem interseções entre expressões e comandos:
  - LPs orientadas a expressão (ALGOL-68 e ML);
  - C é orientada a expressão se considerar fluxo de controle retornando void.

http://nemo.inf.ufes.br/