Linguagens de Imperativa – Parte 2: comandos e blocos

Prof. Arnaldo Candido Junior

UNESP – IBILCE

São José do Rio Preto, SP

Expressões

- **Expressões** são uma das formas mais elementares de computação.
  - Exemplo: expressões aritméticas
- Trabalham com o conceito de **operadores** (ex. soma, divisão, potência, etc)
- Operadores operam sobre **operandos** (literais, variáveis, retornos de função, entre outros)

Operadores

- **Operador unário**: opera sobre um único operando. Exemplo (complemento numérico):
  x = - y;
- **Operador binário**: opera sobre dois operandos. Exemplo (subtração):
  x = 5 - 3;
- **Operador ternário**: opera sobre três operandos. Exemplo (if ternário):
  x = a > b ? a : b;

Operadores (2)

- **Prefixos**: ocorrem antes do operando
  x = ++y;
- **Infixos** (mais comuns): ocorrem entre operandos
  x = 3 + y;
- **Posfixos**: ocorrem após operandos
  x = y++;

Operadores (3)

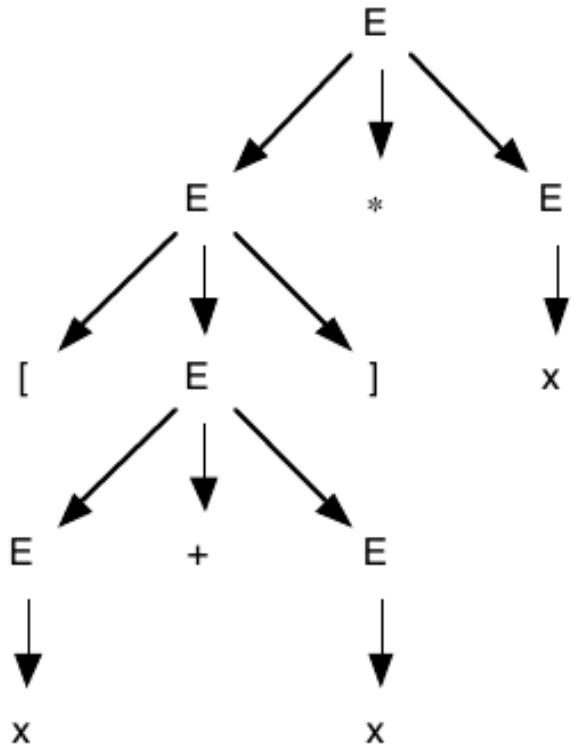
Precedence	Type	Operators	Associativity
1	Postfix	() [] -> . ++ - -	Left to right
2	Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
3	Multiplicative	* / %	Left to right
4	Additive	+ -	Left to right
5	Shift	<<, >>	Left to right
6	Relational	< <= > >=	Left to right
7	Equality	== !=	Left to right
8	Bitwise AND	&	Left to right
9	Bitwise XOR	^	Left to right
10	Bitwise OR	|	Left to right
11	Logical AND	&&	Left to right
12	Logical OR	||	Left to right
13	Conditional	?:	Right to left
14	Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
15	Comma	,	Left to right

Operadores (4)

- **Precedência**: definem quais operadores são analisados primeiro
- **Associatividade**: define se operandos são combinados da esquerda para direita ou vice-versa

Avançado

- Gramática de expressões:
  1. E → E + E
  2. E → E * E
  3. E → (E)
  4. E → x



Avançado (2)

- A gramática anterior é ambígua: existem formas diferentes de criar a mesma expressão
- Aqui está uma versão não ambígua:
  1. E → E + T
  2. E → T
  3. T → T * F
  4. T → F
  5. F → x
  6. F → (E)

Avançado (3)

- Gramáticas de expressões **não caem** na avaliação
  - Os slides trouxeram apenas uma breve explicação sobre elas
  - Gramáticas precisam de ajustes conforme o tipo de analisador sintático usado
  - Mais sobre o tema em linguagens formais e autômatos e compiladores

Efeitos colaterais

- Considere o código do slide a seguir
- O que é exibido ao usuário?

Efeitos colaterais (2)

```c
#include <stdio.h>
int A() {
  printf("A\n");
  return 0;
}
int B() {
  printf("B\n");
  return 0;
}
void main() {
  if (A() && B()) {
    printf("V\n");
  } else {
    printf("F\n");
  }
}
```

Efeitos colaterais (3)

- Operadores em **curto-circuito**: um tipo de otimização feito compilador
  - Efetuada mesmo com -O0
- O código também traz um exemplo de **efeito colateral**
  - A seguir...

Efeitos colaterais (4)

- Um efeito colateral ocorre quando uma expressão é avaliada e o estado do sistema muda
  - Mais comum: variáveis mudam de valor
  - Outros casos: entrada e saída de modo geral (disco, rede, periféricos)

Efeitos colaterais (5)

- Efeitos colaterais são propensos a erros. Eles são proibidos em linguagens puramente funcionais
- Continuam necessários em linguagens procedurais e orientadas a objetos
  - Linguagens modernas trazem recursos para reduzir efeitos colaterais (variáveis imutáveis)
  - Rust: varáveis são imutáveis por padrão

Efeitos colaterais (6)

- Transparência referencial: quando uma expressão pode ser substituída por outra equivalente
  - result1 = (f(a) + b) / (f(a) – c);
  - temp = f(a);
    result2 = (temp + b) / (temp – c);

Efeitos colaterais (7)

- Suponha que f tenha efeitos colaterais
- Suponha ainda, que o programador está contando com esses efeitos
- Compilador pode otimizar expressões
- Levando a surpresas indesejadas
- Conclusão: depender de efeitos colaterais não é uma boa prática

Efeitos colaterais (8)

- Curiosidade: protocolo HTTP foi planejado para lidar efeitos colaterais
  - Get: pensado para não ter efeitos colaterais
  - Update, Delete: efeitos idempotentes (execução repetida não traz efeitos adicionais)
  - Post: efeitos que não são idempotentes
  - Obs: muitas aplicações modernas não seguem a semântica original

Sobrecarga de operadores

- Um operador sobrecarregado tem mais de uma função definida
- Em Java: operador ‘+’ realiza soma (tipos numéricos) ou concatenação (tipos strings)
- Em Python: usuário pode definir sobrecarga para seus próprios tipos
  - A seguir...

Sobrecarga de operadores (2)

```
class Fracao:
  def __init__(self, n, d):
    self.n = n
    self.d = d
  def __add__(self, f2):
    f3 = Fracao(0, 0)
    f3.n = self.n*f2.d + f2.n*self.d
    f3.d = self.d + f2.d
    return f3
  def __mul__(self, f2):
    f3 = Fracao(self.n*f2.n, self.d*f2.d)
    return f3
  def __str__(self):
    return str(self.n) + "/" + str(self.d)
```

Sobrecarga de operadores (3)

```
x = Fracao(1, 2)
y = Fracao(3, 4)
z = x + y
w = x * y
print(z)
print(w)
```

Sobrecarga de operadores (3)

- Algumas linguagens também permitem sobrecarga de funções
- Por exemplo, em Java, a função System.out.println pode ser usada com ints, strings, booleans, etc

Conversão de tipos

- Implícita
  - Coerção: em tempo de compilação nas linguagens estaticamente tipadas
  - Em tempo de execução nas linguagens dinamicamente tipadas
- Explícita:
  - Em C, usando cast
    int x = (int) 3.14;

Erros em expressões

- Expressões aritméticas estão sujeitas a erros de overflow e underflow
  - Devido ao custo, muitas linguagens optam por não tratar overflow e underflow
  - Mesmo linguagens mais focadas em segurança
- Outros erros incluem divisão por zero
- Linguagens como Java, C# e Python tratam erros por meio de **exceções**

Atribuição

- Atribuição simples
  - Variável à esquerda (LHS – Left Hand Side)
  - Seguida de operador de atribuição
  - Seguida de expressão à direita (RHS – Right Rand Side)
- Exemplo: x = y + 3;

Atribuição (2)

- Atribuições múltiplas (ex.: Python)
  (x, y) = (y, x)
- Atribuição condicional (ex.: Perl)
  ($flag ? $count1 : $count2) = 0;
- Atribuição em estruturas de controle de fluxo (ex.: C):
  while ((ch = getchar()) != EOF) { ... }

Atribuição (3)

- Deestruturação: caso especial de atribuição múltipla
  - Intuitivamente, deestruturar significa quebrar uma estrutura de dados em estruturas menores
  - Exemplo Rust:
    let tripla = (0, -2, 3);
    let (a, b, c) = tripla;

Estruturas de Controle de Fluxo

- Controlam a ordem de execução dos comandos
- Tipos de estruturas
  - Estruturas de seleção (ou estruturas condicionais)
  - Estruturas de repetição (ou laços)
  - Outras: por exemplo, desvio não condicional (go to)

Seleção

- Seleção simples: “if” e afins
  - Com ou sem “else”
- Seleção encadeada “if” seguido de um ou mais else “if’s”
  - Em Python: implementado com “if”, “elif” e “else”
- Seleção múltipla: “switch case” e afins

Seleção (2)

- Estrutura geral:

```
se expressao_controle
   entao clausula1
   senao clausula2
```

Seleção (3)

- Qual mensagem o código C abaixo exibe?

```
if (3 < 4)
  printf("A");
  if (1 >= 6)
    printf("B");
else
  printf("C");
```

Seleção (4)

- A gramática que gera o “else” na linguagem C é **ambígua**
- O código anterior pode ser interpretado de duas formas diferentes
- A decisão da linguagem C é que um “else” sempre relaciona-se ao “if” mais próximo sem “else”
- Ambiguidade pode levar a confusão. O código anterior está com a identação incorreta

Seleção: variações

- A linguagem Perl oferece uma estrutura chamada “unless”, equivalente a um “if not”
  - O comando:
    - unless ($x < 3) { print("msg"); }
  - É equivalente a:
    - if (! $x < 3) { print("msg"); }

Seleção: variações (2)

- Perl também permite mudar a ordem entre o “if” e o comando:
  - $x = 4;
    print("maior que três") if ($x > 4);

Seleção: variações (3)

- Em shell script, os comandos “&&” e “||” abaixo se comportam como caso especial da selação:
  - cmd1 && cmd2: se cmd1 foi bem sucedido então executar cmd2
  - cmd1 || cmd2: se cmd1 não foi bem sucedido então executar cmd2

Seleção: variações (4)

- Guarded commands: intuitivamente, é uma sequência de if's que pode ser executada em qualquer ordem
- É uma forma mais justa de distribuir processamento entre diferentes threads
- Em Ada, um comando de seleção próprio. Em linguagens modernas, implementado por meio de bibliotecas
- Exemplo teórico a seguir (proposto por Dijkstra)

Seleção: variações (5)

- Exemplo para ordenar um vetor

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

- O exemplo é uma forma concisa e elegante de ordenar um vetor pequeno

Laços

- Existem duas grandes categorias de estruturas de repetição
  - Controlado por contador (ex.: for)
  - Controlado logicamente (ex.: while, do while)
  - Obs: o “for” da linguagem C é híbrido (controlado contador ou logicamente)

Laços (2)

- Controlado por contador. Exemplos em Kotlin:

```
for (i in 1..10) print(i)
for (i in 10 downTo 1) print(i)
for (i in 1..10 step 2) print(i)
```

- Controlado por contador com “for each”. Exemplos em Python:

```
for cor in ["verde", "azul", "laranja"]:
for i, cor in enumerate(["a", "b", "c"]):
# obs.: também funciona para dicionários
```

Laços (3)

- **while**: controlado logicamente com condição no começo do laço.
  - Permanece no laço enquanto condição é verdadeira
- **do while**: controlado logicamente com condição no final do laço

Laços: controle manual

- Controle manual de laços
  - **continue**: retorna ao começo do laço em questão
  - **break**: sai imediatamento do laço em questão
  - Exemplos vistos em disciplina anterior
- Laços nomeados: um forma prática de usar os comandos em laços aninhados.

Laços: controle manual (2)

- Laços nomeados em Java:

```
externo:
for (int i = 0; i < 10; i++) {
  interno:
  for (int j = 0; j < 10; i++) {
    System.out.println(i*j);
    if (i*j == 49) {
      break externo;
    }
  }
}
```

Laços: variações

- “repeat until” em Pascal, Delphi: semelhante ao “do while”
  - Mas permanece no laço enquanto condição é falsa
- List comprehensions em Python: laços para criação de listas

```
lista = [10, 11, 12, 13, 14, 15]
quadrados = [item*item for item in lista]
```

Laços vs Linguagem Funcional

- Linguagens puramente funcionais não possuem laços, mas eles podem ser emulados com a seguinte estratégia:
  - Corpo do laço é armazenado em uma primeira função
  - Uma segunda função recebe o contador e o corpo, executando o corpo repetidas vezes
  - Exemplo a seguir em Lisp

Laços vs Linguagem Funcional (2)

```
(defun exibe-mensagem (msg)
  (format t "~a~%" msg))

(defun meu-laco (contador corpo)
  (if (= contador 0)
      (funcall corpo contador)
      (progn
        (funcall corpo contador)
        (meu-laco (- contador 1) corpo))))
```

Laços vs Linguagem Funcional (3)

```
; chamada tradicional
(meu-laco 10 #'exibe-mensagem)

; chamada mais elegante com funções anônimas
; mais sobre o assunto na continuação deste material
(meu-laco 10 (lambda (msg)
               (format t "~a~%" msg)))
```

Laços: boas práticas

- Boas práticas:
  - Não se altera manualmente a variável sobre a qual um laço “for” itera
  - Não se altera a coleção (array, dicionário, etc) sobre o qual um laço “for each” itera
  - Algumas linguagens inclusive proíbem essas alterações (ex.: Rust)

Desvio não condicional

- O uso do comando “goto” via de regra não é uma boa prática
- Mas existem casos **pontuais** em que ainda pode ser usado
- Devido a isso, pode ser encontrado mesmo em algumas linguagens modernas (ex.: C#)

Desvio não condicional (2)

- Exemplos de uso: autômatos finitos; switch case; quebra de laços (linguagens sem laços nomeados)
- Importante: idealmente, um goto não deve pular de uma função para outra
  - Pode corromper a stack se isso for feito