Paradigmas de Linguagens de Programação

Expressões e sentenças de atribuição

Introdução

- Expressões sãos o **meio fundamentais** de especificar computações em uma linguagem de programação.
- Para entendê-las, é necessário compreender tanto a **sintaxe** quanto a **semântica** das expressões.
- A avaliação de expressões envolve:
  - **Ordem de avaliação de operadores** (associatividade e precedência).
  - **Ordem de avaliação de operandos**, que pode variar entre implementações.
- Diferenças na ordem de avaliação podem levar a **resultados distintos em programas**.

Introdução

- A essência das linguagens **imperativas** é o papel dominante das **sentenças de atribuição**.
- A **finalidade** dessas sentenças é causar o **efeito colateral** de alterar os valores de variáveis, ou o estado do programa.
- Assim, um conceito central nessas linguagens é o de **variáveis cujos valores mudam durante a execução**.
- **Em linguagens funcionais**
  - Variáveis representam **parâmetros de funções**.
  - Sentenças de declaração vinculam nomes a valores, mas **não produzem efeitos colaterais**.

Expressões Aritméticas

- Um dos principais objetivos das primeiras linguagens de alto nível foi permitir a avaliação automática de expressões aritméticas.
- Características herdadas da matemática:
  - Operadores, operandos, parênteses e chamadas de funções.
  - Operadores podem ser:
    - **Unário**: 1 operando
    - **Binário**: 2 operandos
    - **Ternário**: 3 operandos

Expressões Aritméticas

- **Unário:** 1 operando
  - Ex: <operador><operando>
  - *-x (negação), !flag (NOT lógico)*
- **Binário:** 2 operandos
  - Ex: <operando><operador><operando>
  - *a + b, x * y, p && q*
- **Ternário:** 3 operandos
  - Ex: <operando1> ? <operando2> : <operando3>
  - *cond ? x : y*

Expressões Aritméticas

- Na maioria das linguagens, operadores binários usam **notação infixa** (entre os operandos).
  - **Ex (infix):** a + b
- **Exceção:** Perl (e notação polonesa), que também permite **operadores prefixados** (antes dos operandos).
  - **Ex (prefix):** + a b
- Objetivo das expressões aritméticas: **especificar uma computação aritmética**.
- Implementação envolve:
  - Obter operandos (normalmente da memória).
  - Executar operações aritméticas sobre eles.

Expressões Aritméticas

Questões de projeto:

- Quais são as regras de **precedência** de operadores?
- Quais são as regras de **associatividade**?
- Qual é a **ordem de avaliação dos operandos**?
- Existem **restrições sobre efeitos colaterais** na avaliação?
- A linguagem permite **sobrecarga de operadores** definida pelo usuário?
- Que tipo de **mistura de tipos** é permitida nas expressões?

Ordem de Avaliação de Operadores

- O valor de uma expressão depende da ordem em que os operadores são avaliados.
- Ex: a + b * c

  Se a = 3, b = 4, c = 5:

  Avaliação **da esquerda para a direita**: (a + b) * c = 35

  Avaliação **da direita para a esquerda**: a + (b * c) = 23
- Para evitar ambiguidade, linguagens de programação seguem **regras de precedência** semelhantes às da matemática.

Ordem de Avaliação de Operadores

- As regras de precedência definem a ordem de avaliação de operadores de **diferentes níveis**.
- Baseadas na hierarquia matemática:
  - Parênteses têm maior prioridade.
  - Multiplicação e divisão têm precedência maior que adição e subtração.
  - Muitas linguagens adicionam operadores específicos:
    - Exponenciação ** (quando disponível).
    - Operadores unários (-).

Ordem de Avaliação de Operadores

- Expressão: a + (-b) * c
  - **válido**
- Expressão: a + - b * c
  - normalmente **inválido**
- Exemplos adicionais:
  - -a / b
  - -a * b
  - -a ** b
    - aqui a ordem de avaliação importa (exponenciação tem precedência mais alta que subtração unária).

Ordem de Avaliação de Operadores

Precendência de operadores

	Ruby	*Linguagens baseadas em C*
Mais alta	**	++ e −− posfixados
	+ e − unários	++ e −− prefixados, + e − unários
	*, /, %	*, /, %
Mais baixa	+ e − binários	+ e − binários

Ordem de Avaliação de Operadores

Precendência de operadores

Exemplo: operadores prefixados e posfixados em C

int x = 3;

int y = x++ + ++x;

- x++ usa o valor atual (3) e depois incrementa: x = 4
- ++x incrementa antes de usar: x = 5
- y = 3 + 5 = 8
- O operador **posfixado (x++)** incrementa **depois** de usar o valor.
- O operador **prefixado (++x)** incrementa **antes** de usar o valor.
- O **posfixado tem precedência maior**, por isso é avaliado primeiro quando aparecem juntos.

Ordem de Avaliação de Operadores

- Quando uma expressão contém duas ocorrências adjacentes de operadores com o mesmo nível de precedência, a ordem de avaliação é definida pela **associatividade** da linguagem.
- Um operador pode ser:
  - **Associativo à esquerda**: avalia primeiro o operador mais à esquerda.
  - **Associativo à direita**: avalia primeiro o operador mais à direita.
- Exemplo em Java:
  - a - b + c → avalia da esquerda para a direita.
- Exemplo em Fortran e Ruby:
  - a ** b ** c → avalia da direita para a esquerda.

Ordem de Avaliação de Operadores

- A maioria das linguagens adota **associatividade da esquerda para a direita**.
- Exceções:
  - **Exponenciação** (**) associativa à direita (Fortran, Ruby).
  - Em algumas linguagens (ex. Visual Basic), exponenciação (^) é associativa à esquerda.

Ordem de Avaliação de Operadores

- As regras de associatividade para algumas linguagens bastante utilizadas são:

Linguagem	*Regra de associatividade*
Ruby	Esquerda: *, /, +, –
	Direita: **
Linguagens baseadas em C	Esquerda: *, /, %, + binário, – binário
	Direita: ++,––, – unário, + unário

- **APL**
  - Todos os operadores têm a mesma precedência.
  - Ordem é sempre da direita para a esquerda.

Ordem de Avaliação de Operadores

- Muitas operações aritméticas são **matematicamente associativas**, mas em computação isso pode não se manter.
- Exemplos:
  - Operações de **ponto flutuante** → dependem de precisão limitada → não são estritamente associativas.
  - *(a + b) + c* pode não ser igual a *a + (b + c)* por causa de **overflow** ou **erros de arredondamento**.
- Compiladores podem **reordenar operações associativas** para otimização.
- Programadores podem evitar problemas **especificando a ordem com parênteses**.

Ordem de Avaliação de Operadores

- Parênteses **alteram as regras de precedência e associatividade**.
- Exemplo:
  - (a + b) * c → soma avaliada antes da multiplicação.
  - (a + b) + (c + d) → pode evitar **transbordamento**.
- Uso de parênteses torna o código **mais claro**, sem depender da memorização das regras.

Ordem de Avaliação de Operadores

- **Vantagens**: clareza, controle explícito da ordem de avaliação.
- **Desvantagens**: excesso de parênteses pode deixar o código **verboso** e menos legível.
- Em algumas linguagens, como **APL**, toda ordem de avaliação deve ser definida com parênteses (decisão do projetista Ken Iverson).

Ordem de Avaliação de Operadores

- Ruby é uma linguagem **orientada a objetos pura**: todos os valores (incluindo literais) são objetos.
- Todos os operadores aritméticos, relacionais, de atribuição, indexação, deslocamentos e lógicos bit a bit são implementados como **métodos**.
- Exemplo:
  - a + b → é equivalente a uma chamada do método + no objeto a, com b como parâmetro.
- **Consequência**: operadores podem ser **sobrescritos (redefinidos)** por programas de aplicação.
- Isso permite **sobrecarga de operadores** definida pelo usuário.

Ordem de Avaliação de Operadores

- Assim como em Ruby, as operações aritméticas e lógicas em Lisp são implementadas como subprogramas.
- Diferença: em Lisp os subprogramas precisam ser chamados **explicitamente**.
- Exemplo da expressão a + b * c:
  - Em Lisp → (+ a (* b c))
  - Aqui, + e * são simplesmente os **nomes de funções**.

Ordem de Avaliação de Operadores

Expressões Condicionais

- Permitem atribuir valores com base em uma **condição booleana**.
- Exemplo em C (forma tradicional):

```
if (count == 0)
    average = 0;
else
    average = sum / count;
```

- Forma mais compacta usando **operador condicional (ternário)**:

```
average = (count == 0) ? 0 : sum / count;
```

Ordem de Avaliação de Operadores

- Sintaxe geral:

expressão_1 ? expressão_2 : expressão_3

- expressão_1: condição booleana.
- Se verdadeira o resultado é expressão_2.
- Se falsa o resultado é expressão_3.
- O operador ? : é **ternário**.
- Presente em várias linguagens baseadas em C, além de **Perl, JavaScript e Ruby**.
- Pode ser usado **em qualquer contexto de expressão**.

Ordem de avaliação de **Operandos**

- **Variáveis:** avaliadas buscando seus valores na memória.
- **Constantes:** podem ser buscadas na memória ou já estarem na própria instrução da linguagem de máquina.
- **Expressões entre parênteses:** se um operando é uma expressão entre parênteses, todos os operadores que ela contém devem ser avaliados antes que seu valor possa ser usado como operando.

```
x = (a + b) * c;
```

Ordem de avaliação de Operandos

- Se os operandos **não produzem efeitos colaterais**, a ordem de avaliação é irrelevante.
- O caso interessante ocorre quando um **operando tem efeito colateral** (ex: chamada de função com atualização de variável global).
- A ordem de avaliação passa a impactar o **resultado final do programa**.

Ordem de avaliação de Operandos

Efeitos Colaterais

- **Efeito colateral funcional** ocorre quando uma função:
  - Modifica um **parâmetro passado por referência**;
  - Ou altera uma **variável global**.
  - Exemplo onde *fun(a)* retorna 10 e muda *a* para 20

```
a = 10;
b = a + fun(a);  // se fun altera a
```

O resultado depende da **ordem de avaliação** de *a* e *fun(a)*.

Se *a* for avaliado **primeiro**: b = 20

Se fun(a) for avaliado **primeiro**: b = 30

Ordem de avaliação de Operandos

- Exemplo em C

  O valor de b pode ser:

  8 se a for avaliado primeiro.

  20 se fun1() for avaliado primeiro.

- Mostra como a ordem de avaliação gera **resultados diferentes**.

```
int a = 5;
int fun1() {
   a = 17;
   return 3;
}
void main() {
   int b = a + fun1();
}
```

Ordem de avaliação de Operandos

- Esse efeito gera **comportamento imprevisível**.
- Quebra a ideia de que expressões deveriam ser avaliadas sem efeitos externos.
- Complica a **otimização de compiladores**, que podem reordenar instruções.
- Em **matemática pura** não há efeitos colaterais, pois não existem variáveis.

Ordem de avaliação de Operandos

- **Proibir efeitos colaterais funcionais**
  - Sem parâmetros de duas vias (passagem por referência).
  - Sem acesso a variáveis globais em funções.
  - Vantagem: elimina o problema.
  - Desvantagem: reduz flexibilidade (ex.: C e C++ usam globais por eficiência).
- **Impor ordem fixa de avaliação**
  - Define sempre a mesma ordem para os operandos.
  - Exemplo: Java avalia da esquerda para a direita.
  - Vantagem: previsibilidade.
  - Desvantagem: restringe otimizações de compiladores.

Ordem de avaliação de Operandos

- Não há solução perfeita.
- Cada linguagem equilibra **eficiência** e **segurança**:
  - C e C++ permitem efeitos colaterais, priorizando eficiência.
  - Java impõe ordem fixa, priorizando previsibilidade.
- Programadores devem estar atentos, pois a ordem de avaliação pode alterar o resultado de uma expressão com efeitos colaterais.

Operadores Sobrecarregados

- **Sobrecarga de operadores**: uso de um mesmo operador para mais de um propósito.
- Exemplo comum:
  - **+** → soma de inteiros e ponto flutuante.
  - Em Java, também usado para **concatenação de strings**.
- Aceitável desde que **clareza e confiabilidade** não sejam comprometidas.

Operadores Sobrecarregados

- Uso múltiplo pode afetar **legibilidade** e **detecção de erros** pelo compilador.
- Exemplo em C++: operador &
  - Binário: operação lógica bit a bit (AND).
  - Unário: operador de endereço (&x).
- Problema: omitir um operando pode transformar a operação em unária sem erro imediato e se torna difícil de diagnosticar.
- Situação parecida ocorre com o operador **-** (subtração vs. unário negativo).

Operadores Sobrecarregados

- Linguagens como **C++, C#, F#, Ada** permitem que o programador defina sobrecarga de operadores.
- Exemplo: definir * entre uma matriz e um escalar significa multiplicar todos os elementos da matriz pelo escalar.
- Vantagens:
  - Pode **melhorar a legibilidade** (ex.: A * B + C * D em vez de chamadas de função).
- Riscos:
  - Usuário pode definir operações sem sentido.
  - Diferentes módulos podem sobrecarregar o mesmo operador de formas distintas gerando confusão.

Operadores Sobrecarregados

- A sobrecarga definida pelo usuário deve ser usada **com bom senso**.
- Vantagens: código mais legível e expressivo.
- Desvantagens: perda de legibilidade, risco de ambiguidade.
- Restrições: alguns operadores **não podem ser sobrecarregados** (**.** e **::** em C++).
- Observação histórica: Java inicialmente não incluiu sobrecarga de operadores, mas C# reintroduziu o recurso.

Conversões de Tipos

- **Conversão de estreitamento (narrowing)**
  - Converte para um tipo que **não pode representar todos os valores** do tipo original.
  - Exemplo: double → float, float → int.
  - Pode ser **insegura**, alterando a magnitude ou descartando informação.
- **Conversão de alargamento (widening)**
  - Converte para um tipo que pode representar **ao menos aproximações de todos os valores** do tipo original.
  - Exemplo: int → float.
  - Normalmente mais **segura**, mas pode causar perda de precisão.

Conversões de Tipos

- Conversão de **estreitamento** em Java:

```
double x = 1.3E25;
int y = (int) x; // valor não se relaciona ao original
```

- Conversão de **alargamento** também pode perder precisão:
- Inteiros de 32 bits → ponto flutuante de 32 bits (apenas ~7 dígitos decimais).
- Possível perda de 2 dígitos de precisão.

Conversões de Tipos

- Conversões de tipos não primitivos (ex.: matrizes, registros, classes) são mais complexas.
- Conversões podem ser:
  - **Explícitas (cast)** → programador indica a transformação.
  - **Implícitas (coerção)** → feitas automaticamente pela linguagem.

Conversões de Tipos

Expressões de Modo Misto

- Ocorrem quando os **operandos têm tipos diferentes**.
- Exemplo em Java:

```
int a;
float b, c, d;
d = b * a;   // int convertido automaticamente em float
```

- A conversão implícita é chamada de **coerção**.
- Objetivo: permitir operações entre tipos diferentes sem erro de compilação.

Conversões de Tipos

- **Vantagem**
  - Facilita a programação, evitando casts explícitos.
- **Desvantagem**
  - Reduz a **capacidade de detecção de erros de tipo** pelo compilador.
  - Um erro de digitação (ex: usar variável errada) pode passar despercebido, pois o compilador insere coerção automática.

Conversões de Tipos

- **C e C++**: tipos menores que int (char, short) são sempre promovidos para int em expressões.
- **Java**: tipos byte e short sofrem coerção para int em operações aritméticas.
- **F#, ML, Ada**: **não permitem coerção implícita** em expressões, aumentando a segurança de tipos.
- Outras linguagens permitem coerções numéricas amplas (ex: int → float).

Conversões de Tipos

Conversão de Tipo Explícita

- A maioria das linguagens permite conversões **explícitas** (**estreitamento** ou **alargamento**).
- Em linguagens baseadas em C, chamadas de **casts**:

  (int) angle

- Em ML e F#, usam sintaxe de chamada de função:

  float(sum)

Usadas quando o programador precisa **forçar** a conversão.

Conversões de Tipos

- Linguagem **PL/I** adotou coerções muito flexíveis.
- Permitido: combinar praticamente qualquer tipo de dado em expressões.
- Exemplo:
  - Uma variável do tipo cadeia de caracteres pode ser o operando de um operador aritmético, com um inteiro como o outro operando.
  - String podia ser varrida em tempo de execução em busca de número.
  - Se encontrasse, convertia para float automaticamente.
- Problema:
  - **Verificação de tipos adiada** para tempo de execução.
  - Difícil detectar erros de programação gerando perda de confiabilidade.

Conversões de Tipos

- **Erros em expressões** podem ocorrer devido a limitações da aritmética computacional:
  - **Transbordamento (overflow):** resultado maior que a capacidade do tipo.
  - **Subtransbordamento (underflow):** resultado muito pequeno para ser representado.
  - **Divisão por zero:** matematicamente indefinida, mas possível em programas.
- Muitas linguagens não tratam esses erros automaticamente.
- Algumas oferecem **exceções** para detectar e tratar esses casos.

Expressões Relacionais e Booleanas

Expressões Relacionais

- Um **operador relacional** compara os valores de dois operandos.
- Uma **expressão relacional** tem dois operandos e um operador relacional.
- O valor da expressão é **booleano** (verdadeiro/falso), exceto quando a linguagem não tem tipo booleano específico.
- Operadores relacionais são sobrecarregados em muitos tipos: numéricos, cadeias de caracteres, enumerações.
- Exemplo simples: comparação entre inteiros (a < b).
- Exemplo complexo: comparação de strings ("abc" < "xyz").

Expressões Relacionais e Booleanas

- A sintaxe varia entre linguagens:
  - C, C++: !=
  - Lua: ~=, Fortran 95+: .NE.
  - ML, F#: <>
- JavaScript e PHP possuem também:
  - === (igualdade sem coerção)
  - !== (desigualdade sem coerção)
- Ruby:
  - == (com coerção).
  - eql? (sem coerção).
  - === (usado em cláusulas case).

Expressões Relacionais e Booleanas

- Operadores relacionais têm **precedência menor** que os aritméticos.
- Em a + 1 > 2 * b, a soma e multiplicação são avaliadas antes da comparação.
- Garantia de consistência: primeiro operações matemáticas, depois comparações.

Expressões Relacionais e Booleanas

- **Fortran I (anos 50)** usava abreviações em inglês para operadores relacionais.
- Motivo: perfuradoras de cartões não tinham os símbolos < e >.
- Exemplos:

  .GT. (greater than), .LT. (less than).

- Essa decisão influenciou a sintaxe das primeiras linguagens.

Expressões Relacionais e Booleanas

Expressões Booleanas

- Consistem em:
  - Variáveis e constantes booleanas.
  - Expressões relacionais.
  - Operadores booleanos (AND, OR, NOT, às vezes XOR e equivalência).
- Operadores booleanos recebem operandos booleanos e produzem valor booleano.
- Nas linguagens baseadas em C, **&&** (AND) tem precedência maior que || (OR).

Expressões Relacionais e Booleanas

- A precedência dos operadores aritméticos, relacionais e booleanos nas linguagens baseadas em C é a seguinte:

Mais alta	++ e – posfixados
	+ unário, – unário, ++, –– e ! prefixados
	*, /, %
	+ binário, – binário
	<, >, <=, >=
	=, !=
	&&
Mais baixa	||

Expressões Relacionais e Booleanas

- Em linguagens como **Perl** e **Ruby**, existem também versões por extenso:
  - and, or (com menor precedência).
- Observação: operadores booleanos têm **precedência menor que os relacionais**, que por sua vez têm precedência menor que os aritméticos.

Expressões Relacionais e Booleanas

- Em versões de C anteriores ao C99:
  - Não havia tipo booleano: inteiros usados como booleanos.
    - Zero (0): falso
    - Qualquer valor diferente de zero: verdadeiro.
- Exemplo incomum em C:

```
a > b > c   // válido
```

- Avaliação: (a > b) resulta em 0 ou 1.
- Esse resultado é comparado com c. (b nunca é comparado com c)

Avaliação em Curto-Circuito

- Uma **avaliação em curto-circuito** ocorre quando o valor de uma expressão pode ser determinado **sem avaliar todos os operandos**.
- Exemplo aritmético:
  - (13 * a) * (b / 13 - 1)
- Se a = 0, não há necessidade de avaliar (b/13 - 1).
- Exemplo booleano:
  - (a > 0) && (b < 10)
- Se a > 0 for falso, não é necessário avaliar (b < 10).

Avaliação em Curto-Circuito

- Sem curto-circuito, **todos os operandos são sempre avaliados**.
- Pode causar erros de execução.

```
index = 0;
while ((index < listlen) && (list[index] != key))
   index++;
```

Se não houver curto-circuito, list[index] pode gerar erro de acesso fora dos limites.

Avaliação em Curto-Circuito

- **C, C++, Java**: operadores && e || usam curto-circuito.
  - Mas também existem & e | bit a bit, que **não** são avaliados em curto-circuito.
- **Ada**: programador pode escolher:
  - and then, or else → com curto-circuito.
  - and, or → sem curto-circuito.
- **Ruby, Perl, ML, F#, Python**: todos os operadores lógicos são avaliados em curto-circuito.

Sentenças de Atribuição

Atribuições Simples

- Forma mais comum de atribuição:

  <variável> <operador de atribuição> <expressão>

- Diferenças entre linguagens:
- = em C, Fortran, BASIC.
- := em ALGOL, Pascal, Ada → evita confusão com igualdade relacional.
- Exemplo:

```
x = y + z;      // C
x := y + z;     // Pascal
```

Sentenças de Atribuição

Alvos Condicionais

- Algumas linguagens permitem **especificar o alvo da atribuição condicionalmente**.
- Exemplo em Perl:

```
($flag ? $count1 : $count2) = 0;
```

- Equivalente a:

```
if ($flag) {
   $count1 = 0;
} else {
   $count2 = 0;
}
```

Sentenças de Atribuição

Operadores de Atribuição Compostos

- Forma abreviada para expressões que usam a própria variável no cálculo.
- Exemplo:

```
a = a + b;
```

- pode ser escrito como:

```
a += b;
```

- Introduzidos no **ALGOL 68**, depois adotados por C, Java, Perl, Python, Ruby etc.
- Aumentam a legibilidade e reduzem redundância.

Sentenças de Atribuição

Operadores de Atribuição Unários

- Presentes em C, Perl, JavaScript.
- **Unem incremento/decremento com atribuição.**
- Podem ser usados como:
  - **Pré-fixados** (++count): incrementa antes de usar o valor.
  - **Pós-fixados** (count++): usa o valor atual e incrementa depois.
- Exemplos:

```
sum = ++count;   // count incrementado, depois atribuído a sum
sum = count++;   // count atribuído a sum, depois incrementado
```

Sentenças de Atribuição

- count++ é equivalente a:

```
count = count + 1;
```

- Também podem ser combinados com outros operadores unários:

```
-count++   // count incrementado, depois negado
```

- Regra: **quando dois operadores unários aparecem juntos, a associação é da direita para a esquerda.**
- Exemplo:

```
-count++
```

- equivale a

```
-(count++)
```

Sentenças de Atribuição

Atribuição como Expressão

- Em C, C++ e Java, a **atribuição retorna um valor:** o valor atribuído.
- Isso permite usar atribuições dentro de expressões:

```
while ((ch = getchar()) != EOF) {
        ...
}
```

- O operador de atribuição se comporta como qualquer outro operador binário (com efeito colateral).
- Pode permitir **atribuições de múltiplos alvos**:

```
sum = count = 0;
```

Sentenças de Atribuição

- **Baixa legibilidade**: expressões tornam-se difíceis de entender.
- **Erros sutis**:

  if (x = y) ...

  em vez de:

  if (x == y) ...
- O código compila, mas o teste lógico é substituído por atribuição.
- **Java e C#** limitam o problema → exigem que condições em if sejam **booleanas**, evitando que inteiros sejam usados diretamente.
- Conclusão: flexível, mas arriscado → cuidado com erros de digitação e efeitos colaterais.

Sentenças de Atribuição

Atribuições Múltiplas

- Algumas linguagens modernas (Perl, Ruby, Lua, Python) suportam **atribuições de múltiplos alvos e fontes**.
- Exemplo em Perl:

  ($first, $second, $third) = (20, 40, 60);
- 20 → $first, 40 → $second, 60 → $third.
- Permite trocas diretas de variáveis sem temporários:

  ($first, $second) = ($second, $first);
- Ruby e Perl permitem sintaxe ainda mais simples, sem parênteses obrigatórios.
- Em Ruby e Python, também existem formas adicionais de atribuição, como **desestruturação de arrays**.

Sentenças de Atribuição

Atribuições em Linguagens Funcionais

- Em linguagens funcionais, **identificadores não são variáveis mutáveis**, mas **nomes vinculados a valores**.
- Em ML:

```
val cost = quantity * price;
```

- Cria um novo nome cost, sem modificar versões anteriores.
- No F#, usa-se let e val:
  - val cria uma nova associação.
  - let permite **declarações aninhadas**.
- Importante: não existe “reatribuição”, apenas **criação de novos vínculos**.

Sentenças de Atribuição

- **Java e C#**:
  - Só permitem atribuições de alargamento (*widening*).
  - Ex.: int → float permitido, mas float → int não.
  - Aumenta confiabilidade (evita perdas silenciosas).
- **Ada**:
  - Não há coerção automática em atribuições.
- **Funcionais**:
  - Não existe atribuição de modo misto (valores apenas nomeados).

Resumo

- Expressões: compostas por constantes, variáveis, parênteses, chamadas a funções e operadores
- Ordem de avaliação definida por **precedência e associatividade dos operadores**
- Conversões de tipo podem ser:
  - **Alargamento (widening)** → geralmente seguras
  - **Estreitamento (narrowing)** → podem causar perda de dados ou erros
- Conversões podem ser **explícitas (casts)** ou **implícitas (coerções)**

Resumo

- Tipos de erros em expressões:
  - **Overflow / Underflow** (limitações aritméticas)
  - **Divisão por zero**
- Sentenças de atribuição aparecem em várias formas:
  - **Simples** (= ou :=)
  - **Condicionais** (Perl, etc.)
  - **Compostas** (+=, -=, etc.)
  - **Unárias** (++/–)
  - **Múltiplas** (Perl, Ruby, Lua, Python)
- Atribuição mista:
  - **C/C++** → qualquer tipo numérico
  - **Java/C#** → apenas widening
  - **Funcionais** → não existe

Paradigmas de Linguagens de Programação

Expressões e sentenças de atribuição