Paradigmas de Linguagens de Programação

Estruturas de controle no nível de sentença

Introdução

- O **fluxo de controle**, ou sequência de execução, define a ordem em que instruções são executadas em um programa.
- Nas linguagens imperativas, ele ocorre **entre sentenças** (nível de sentença).
- Esse capítulo aborda **sentenças de controle**, responsáveis por escolher e repetir caminhos de execução.
- As formas mais comuns são:
  - **Seleção** (if, switch);
  - **Repetição** (while, for, do);
  - **Desvio incondicional** (goto, break, continue).

Introdução

- Computações em linguagens imperativas ocorrem por meio da **avaliação de expressões** e da **atribuição de valores** a variáveis.
- Para tornar essas computações **flexíveis e eficientes**, surgem mecanismos adicionais:
  - Escolher entre caminhos alternativos;
  - Executar sentenças repetidamente.
- Sentenças que fornecem essas capacidades são chamadas de **sentenças de controle**.

Introdução

- Em **linguagens funcionais**, a execução ocorre pela **avaliação de expressões e funções**.
  - Funções podem ser aplicadas repetidamente a parâmetros, gerando comportamento análogo à repetição.
- Assim, as **sentenças de controle funcionais** são construídas sobre **expressões** e **funções**, não sobre atribuições.
- Exemplo: em Lisp ou ML, a recursão substitui loops como forma de repetição.

Introdução

- As primeiras linguagens bem-sucedidas, como **Fortran**, foram criadas por engenheiros de hardware.
- O foco inicial era em **eficiência e uso de recursos**, não em legibilidade.
- Isso levou ao uso de **instruções de salto (goto)** como meio de controle de fluxo.
- Com o tempo, percebeu-se que o **excesso de saltos** prejudicava a **compreensão e manutenção** do código.

Introdução

- Linguagens modernas exigem sentenças de controle **suficientemente poderosas** para expressar a lógica desejada,
- mas também **simples e legíveis**.
- O número de **formas de controle** deve ser:
  - Grande o bastante para dar flexibilidade;
  - Pequeno o bastante para manter a **clareza** da linguagem.
- **Objetivo do projeto:** encontrar o equilíbrio entre **flexibilidade** e **simplicidade**.

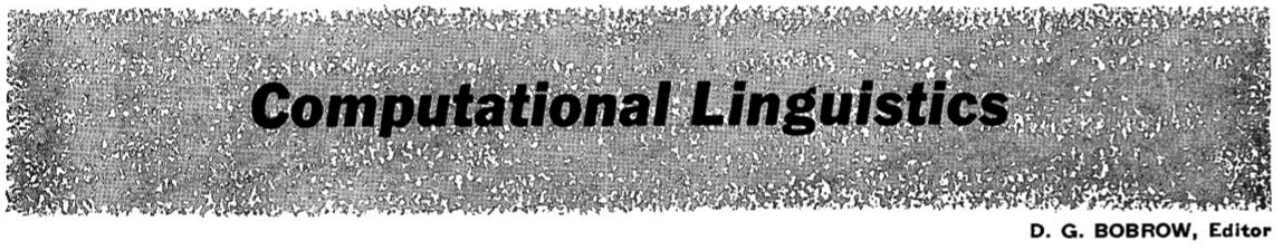
Introdução

- Uma **estrutura de controle** é uma **sentença de controle** e a coleção de sentenças cuja execução ela controla.
- Tipos principais:
  - **Seleção:** escolhe um caminho (if, switch);
  - **Iteração:** repete um bloco (for, while);
  - **Desvio incondicional:** muda o fluxo diretamente (goto, break).
- Cada estrutura possui regras de avaliação e questões de projeto próprias.

Introdução

- “Foi provado que todos os algoritmos que podem ser expressos por diagramas de fluxo podem ser codificados em uma linguagem de programação com apenas **duas** sentenças de controle: uma para escolher entre **dois caminhos de fluxo** de controle e uma para **iterações** logicamente controladas.” Sebesta

Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules. **Corrado Böhm & Giuseppe Jacopini**, *Communications of the ACM* (1966)



Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules

CORRADO BÖHM AND GIUSEPPE JACOPINI
International Computation Centre and Istituto Nazionale per le Applicazioni del Calcolo, Roma, Italy

In the first part of the paper, flow diagrams are introduced to represent *inter al.* mappings of a set into itself. Although not every diagram is decomposable into a finite number of given base diagrams, this becomes true at a semantical level due to a suitable extension of the given set and of the basic mappings defined in it. Two normalization methods of flow diagrams are given. The first has three base diagrams; the second, only two.

In the second part of the paper, the second method is applied to the theory of Turing machines. With every Turing maching provided with a two-way half-tape, there is associated a similar machine, doing essentially the same job, but working on a tape obtained from the first one by interspersing

In this paper, flow diagrams are introduced by the ostensive method; this is done to avoid definitions which certainly would not be of much use. In the first part (written by G. Jacopini), methods of normalization of diagrams are studied, which allow them to be decomposed into base diagrams of three types (first result) or of two types (second result). In the second part of the paper (by C. Böhm), some results of a previous paper are reported [8] and the results of the first part of this paper are then used to prove that every Turing machine is reducible into, or in a determined sense is equivalent to, a program written in a language which admits as formation rules only composition and iteration.

2. Normalization of Flow Diagrams

It is a well-known fact that a flow daigram is suitable for representing programs, computers, Turing machines, etc. Diagrams are usually composed of boxes mutually connected by oriented lines. The boxes are of functional type (see Figure 1) when they represent elementary operations to be carried out on an unspecified object x of a set X, the former of which may be imagined concretely as the set of the digits contained in the memory of a computer, the tape configuration of a Turing machine,

Sentenças de Seleção

- Uma **sentença de seleção** fornece meios para escolher entre dois ou mais caminhos de execução em um programa.
- São **estruturas fundamentais** em todas as linguagens de programação — conceito essencial comprovado por **Böhm e Jacopini (1966)**.
- Podem ser classificadas em:
  - **Seleção de dois caminhos**
  - **Seleção múltipla**

Sentenças de Seleção de dois caminhos

- Forma geral do **seletor de dois caminhos**:

```
if expressão_de_controle
    cláusula então
    cláusula senão
```

- Apesar das semelhanças entre linguagens, há variações de projeto importantes.

Sentenças de Seleção de dois caminhos

Questões de projeto para seletores de dois caminhos:

- Qual **forma e tipo** da expressão que controla a seleção?
- Como são especificadas as **cláusulas então e senão**?
- Como é definido o **significado de seletores aninhados**?

Sentenças de Seleção de dois caminhos

Expressão de Controle

- A **expressão de controle** determina **qual caminho de execução** será seguido.
- É colocada **entre parênteses** quando **a palavra reservada then** (ou outro marcador) **não é usada**.
  - Exemplo: linguagens como **C** exigem parênteses.
- Quando **then** é usado, há **menos necessidade de parênteses**, como em **Ruby**.
- Uso por linguagem:
  - **C89:** sem tipo booleano → expressões **aritméticas** eram usadas como controle.
  - **Python, C99 e C++:** aceitam tanto **expressões aritméticas quanto booleanas**.
  - **Linguagens modernas:** aceitam **apenas expressões booleanas** nas condições.

Sentenças de Seleção de dois caminhos

```
public class TestBoolean {

   public static void main(String[] args) {

            int x = 5;
            if (x) { // Erro de compilação: expressão não é booleana

                        System.out.println("Invalido");

            }
            if (x > 0) { // Correto

                        System.out.println("Valido");

            }
   }
}
```

```
TestBoolean.java:5: error: incompatible types: int cannot be converted to boolean
      if (x) {   // Erro de compilação: expressão não é booleana
           ^
1 error
```

Sentenças de Seleção de dois caminhos

Forma da Cláusula

- Em muitas linguagens, as cláusulas **então** e **senão** podem ser sentenças simples ou compostas.
- Em **Perl**, ambas **devem ser compostas**, mesmo que contenham apenas uma sentença.
- Muitas linguagens usam **chaves {}** para delimitar sentenças compostas.

Sentenças de Seleção de dois caminhos

- Em **Python e Ruby**, as cláusulas **então** e **senão** são **sequências de sentenças**, não sentenças compostas.
  - Em Python, usa-se **indentação** e **dois pontos (:)** em vez de *then*.
- **Exemplo em Python:**

```
if x > y:
   x = y
   print("case 1")
```

- Todas as sentenças endentadas são incluídas na mesma sentença composta.

Sentenças de Seleção de dois caminhos

Aninhamento de Seletores

- Ocorre quando uma sentença if está dentro de outra, e **não fica claro** a qual if a cláusula else pertence.
- Exemplo em Java:
- Essa sentença pode ser interpretada de duas formas:
  - O else pode se associar ao **primeiro if** (externo).
  - Ou ao **segundo if** (interno).
- A maioria das linguagens imperativas resolve associando o else ao **if mais próximo**.

```
if (sum == 0)
   if (count == 0)
      result = 0;
else
      result = 1;
```

Sentenças de Seleção de dois caminhos

- Em **Java**, para forçar a associação alternativa, o if interno deve ser colocado entre chaves:

```
if (sum == 0) {
   if  (count == 0)
      result = 0;
}
else
      result = 1;
```

- • **C, C++ e C#** compartilham o mesmo comportamento.

Sentenças de Seleção de dois caminhos

- Algumas linguagens resolvem o problema de ambiguidade com **palavras reservadas de fechamento**.
- Em **Ruby** e **Lua**, o end marca o final de cada cláusula.
- Exemplo em Ruby (else associado ao if interno):

```
if sum == 0  then
  if count == 0 then
    result = 0
  else
    result = 1
  end
end
```

Sentenças de Seleção de dois caminhos

- Exemplo em Ruby alternativo (else associado ao if externo):

```
if sum == 0  then
  if count == 0 then
    result = 0
  end
else
    result = 1
 end
```

Sentenças de Seleção de dois caminhos

- Exemplo em Ruby alternativo (else associado ao if externo):

```
if sum == 0  then
  if count == 0 then
    result = 0
  end
else
    result = 1
 end
```

Sentenças de Seleção de dois caminhos

- Em **Python**, a **indentação** define os blocos, substituindo o uso de palavras de fechamento:

```
if sum == 0 :
    if count == 0 :
        result = 0
else:
    result = 1
```

- O alinhamento visual deixa claro a qual if o else pertence.
- Linguagens como **ML** não enfrentam o problema, pois **não permitem sentenças if sem cláusula else**.

Sentenças de Seleção múltipla

- A **sentença de seleção múltipla** permite escolher **uma entre várias sentenças ou grupos de sentenças**.
- É uma **generalização** do seletor de dois caminhos.
- Embora seja possível construir um seletor múltiplo a partir de vários *if-else* ou *gotos*, isso resulta em **código ilegível, propenso a erros e difícil de manter**.
- Por isso, há a necessidade de **estruturas específicas** (como switch em C ou case em outras linguagens).

Sentenças de Seleção múltipla

Questões de Projeto

- **Qual é a forma e o tipo da expressão que controla a seleção?**
  - Define o tipo de valor que pode ser testado (ex.: inteiro em C, string em Python, enum em Java).
- **Como são especificados os segmentos selecionáveis?**
  - Refere-se à forma de cada bloco de código associado a um caso. {} (C, Java) ou usar indentação (Python).
- **O fluxo de execução por meio da estrutura pode incluir apenas um segmento selecionável?**
  - Em algumas linguagens, como C, é possível executar vários casos seguidos (*fall-through*).
- **Como os valores de cada caso são especificados?**
  - Define como cada opção é declarada: valores únicos (case 1:), listas (case 1, 2, 3:) ou intervalos (case 1..5:).
- **Como valores da expressão de seleção que não estão representados devem ser manipulados, se é que o devem?**
  - Determina o comportamento quando nenhuma condição é atendida.
  - A solução comum é o uso de um **caso padrão**, como default em C.

Sentenças de Seleção múltipla

- A **sentença switch** em C (também presente em C++, Java e JavaScript) é uma **estrutura de seleção múltipla primitiva**, baseada em **valores constantes discretos**.
- Estrutura geral:

```
switch (expressão) {
   case constante1: sentença1;
   case constante2: sentença2;
   ...
   default: sentençaN;
}
```

Sentenças de Seleção múltipla

```
switch (expressão) {
   case constante1: sentença1;
   case constante2: sentença2;
   ...
   default: sentençaN;
}
```

- A **expressão de controle** deve ser de tipo inteiro, caractere ou enumeração.
- Cada **case** marca um ponto de desvio.
- **default** é opcional, usado quando nenhum valor corresponde.
- Se não houver **break**, o controle **continua no próximo caso** (*fall-through*).

Sentenças de Seleção múltipla

- Exemplo em C
  - **sem break (gera múltiplas execuções)**
  - **Com break (sai após executar um bloco)**

Sentenças de Seleção múltipla

Seleção múltipla com if

- Em muitas situações, uma sentença switch ou case é inadequada para seleção múltipla.
- Quando as seleções devem ser feitas com base em **expressões booleanas**, seletores if aninhados são usados para simular seletores múltiplos.
- Linguagens como **Perl** e **Python** estenderam essa forma para melhorar a legibilidade.
- Essas linguagens introduzem a **cláusula else-if**, chamada elif em Python.

Sentenças de Seleção múltipla

- **Exemplo (Python):**

```
if count < 10:
    bag1 = True
elif count < 100:
    bag2 = True
elif count < 1000:
    bag3 = True
```

Sentenças de Seleção múltipla

- A estrutura elif é semanticamente equivalente a uma sequência de if aninhados:
- A forma elif é mais **legível** e evita o excesso de indentação.
- Cada elif testa uma condição booleana de forma independente.
- Por isso, elif **não é redundante** em relação ao switch.

```
if count < 10:
    bag1 = True
else:
    if count < 100:
        bag2 = True
    else:
        if count < 1000:
            bag3 = True
        else:
            bag4 = True
```

Sentenças de Seleção múltipla

- Algumas linguagens, como **Ruby**, possuem uma forma mais sintética:

```
case
when count < 10 then bag1 = true
when count < 100 then bag2 = true
when count < 1000 then bag3 = true
End
```

- Essa estrutura realiza o mesmo que o if–elif–else.
- A cada when, uma condição booleana é testada de cima para baixo.
- O primeiro when verdadeiro é executado.

Sentenças de Seleção múltipla

- Linguagens funcionais, como **Scheme**, usam a forma especial COND.
- Baseia-se em **expressões condicionais matemáticas**, que avaliam pares (predicado, expressão) sequencialmente.
- A cláusula ELSE é opcional, mas recomendada.
- Cada predicado é avaliado até que um deles seja verdadeiro; sua expressão correspondente é então executada.

```
(COND
 (predicado1,expressão1)
 (predicado2,expressão2)
 ...
 (ELSE expressão_n)
)
```

Sentenças de Seleção múltipla

- Exemplo da expressão COND

```
(COND
  ((> x y) "x is greater than y")
  ((< x y) "y is greater than x")
  (ELSE "x and y are equal")
)
```

- A função COND retorna o valor associado ao primeiro predicado verdadeiro.
- **Exemplo de saída:**
  - Se x = 8 e y = 3 → "x is greater than y"
  - Se x = y → "x and y are equal"

Sentenças de Iteração

- **Uma sentença de iteração** é aquela que faz uma sentença ou uma coleção de sentenças ser executada **nenhuma, uma ou mais vezes**.
- **Uma sentença de iteração é também chamada de laço.**
- A **iteração** é um método de repetição da execução de segmentos de código. **Essência do poder da computação**.
- Se não fosse possível algum modo de execução repetitiva, os programadores teriam de repetir manualmente cada ação.
- Todas as linguagens de programação, desde **Plankalkül**, incluem algum tipo de laço.

Sentenças de Iteração

- As primeiras sentenças iterativas estavam diretamente relacionadas às **matrizes** e aos **cálculos numéricos**.
- O uso de laços se tornou amplamente necessário com o avanço das linguagens e o aumento da complexidade computacional.
- Laços permitem processar grandes quantidades de dados sem repetição explícita de código.

Sentenças de Iteração

Questões de projeto

- Como a iteração é controlada?
- Onde o mecanismo de controle deve aparecer na sentença de laço?

Sentenças de Iteração

- O controle da iteração pode ser feito de várias formas:
  - Por **contadores numéricos** (como em for i = 1 to n)
  - Por **condições lógicas** (como em while ou until)
- A diferença entre as variações está **na posição da verificação da condição**:
  - Se a condição é testada **antes** da execução → **pré-teste**
  - Se a condição é testada **depois** da execução → **pós-teste**
- O **corpo da sentença de iteração** é o conjunto de sentenças que será executado repetidamente.
- O **controle de laço** determina **quantas vezes o corpo será executado** e **quando a execução termina**.

Sentenças de Iteração

Laços Controlados por Contador

- Uma **sentença de controle iterativa de contagem** tem uma variável chamada **variável de laço**, na qual o valor de contagem é mantido.
- Também inclui alguma forma de especificar os valores:
  - **Inicial** e **final** da variável de laço.
  - A diferença entre esses valores, chamada de **tamanho do passo**.
- As especificações de início, fim e passo de um laço são chamadas de **parâmetros do laço**.

Sentenças de Iteração

Laços Controlados por Contador

- Por serem mais complexos, seu **projeto e implementação** exigem mais esforço.
- São especialmente úteis quando se conhece **com antecedência** o número exato de iterações.
- Exemplo típico: o comando **for** em linguagens como C, Java e Python.

Sentenças de Iteração

- **Questões de Projeto**
- **Qual é o tipo e o escopo da variável de laço?**
  - Deve ser uma variável normal (com escopo comum) ou ter escopo especial?
  - Isso afeta o acesso e a legibilidade do código.
- **A variável ou os parâmetros de laço podem ser modificados dentro do laço?**
  - Se sim, como isso influencia o controle da iteração?
  - Modificações internas podem gerar comportamento imprevisível.
- **Os parâmetros de laço devem ser avaliados uma vez ou a cada iteração?**
  - Avaliar uma vez gera laços mais simples e rápidos.
  - Avaliar a cada iteração dá mais flexibilidade, mas aumenta a complexidade.

Sentenças de Iteração

A sentença *for* em C

A forma geral da sentença for de C é:

for (expressão_1; expressão_2; expressão_3)

corpo do laço

- O **corpo** do laço pode ser:
  - uma **única sentença**;
  - uma **sentença composta** (bloco {});
  - ou uma **sentença nula**.
- Cada expressão do for é opcional, se a **segunda expressão** for omitida, ela é considerada **verdadeira**, criando um laço infinito.

Sentenças de Iteração

A sentença *for* em C

Expressão	Função	Avaliação
expressão_1	Inicialização (executada **uma vez** antes do laço)	Antes da 1ª iteração
expressão_2	Condição de controle (termina o laço se for **0 ou falso**)	Antes de cada iteração
expressão_3	Atualização (normalmente incrementa/decrementa a variável de laço)	Após cada iteração

Em C, qualquer valor diferente de zero é considerado **verdadeiro**. O tipo da segunda expressão pode ser **aritmético** ou **booleano** (C99 em diante).

Sentenças de Iteração

A sentença *for* em C

Sebesta descreve o ***for*** de C usando uma forma equivalente em pseudocódigo:

```
    expressão_1
loop:
    if expressão_2 = 0 goto out
    [corpo do laço]
    expressão_3
    goto loop
out: . . .
```

Essa forma mostra que o ***for*** é essencialmente um **laço de pré-teste**, pois a condição (expressão_2) é avaliada antes de cada iteração.

Sentenças de Iteração

A sentença *for* em C

- Exemplo de for em C

```
for (count = 1; count <= 10; count++)
    printf("%d\n", count);
```

- Inicializa count em 1.
- Executa enquanto count <= 10.
- Incrementa count a cada iteração.
- Todas as expressões são opcionais. Se expressão_2 for omitida, o laço será **infinito**.

Sentenças de Iteração

A sentença *for* em C

- Exemplo com múltiplas variáveis

```
for (count1 = 0, count2 = 1.0;
    count1 <= 10 && count2 <= 100.0;
    sum = ++count1 + count2, count2 *= 2.5)
```

- Inicializa duas variáveis (count1 e count2).
- Usa uma condição composta com &&.
- Atualiza duas expressões por vírgula na terceira parte.
- A expressão de atualização pode conter múltiplas sentenças separadas por vírgulas.

Sentenças de Iteração

A sentença *for* em C

- Exemplo com múltiplas variáveis

```
    count1 = 0
    count2 = 1.0
loop:
    if count1 > 10 goto out
    if count2 > 100.0 goto out
    count1 = count1 + 1
    sum  = count1 + count2
    count2 = count2 * 2.5
    goto loop
out: …
```

```
for (count1 = 0, count2 = 1.0;
        count1 <= 10 && count2 <= 100.0;
        sum = ++count1 + count2, count2 *= 2.5)
```

Sentenças de Iteração

A sentença *for* em C

- **C99 / C++** permitem **definir variáveis dentro do for**:

```
for (int count = 0; count < len; count++) {
    ...
}
```

- O escopo da variável é **limitado ao laço**.
- A sentença for do **Java** e **C#** é semelhante, mas a **condição (expressão_2)** deve ser **booleana**.

Sentenças de Iteração

A sentença *for* em Python

- Forma geral:

for *variável_de_laço* ***in*** *objeto****:***
    corpo do laço
*[****else:***
    cláusula senão]

- O **valor do objeto** é atribuído à variável de laço **uma vez por iteração**.
- A **cláusula else** (opcional) é **executada apenas se o laço termina normalmente**, ou seja:
  - **Sem interrupção com break**.
  - **Após percorrer completamente o iterável**.

Sentenças de Iteração

A sentença *for* em Python

- Exemplo básico

```
for count in [2, 4, 6]:
    print(count)
else:
    print("Laço concluído normalmente.")
```

- O **else** é executado porque o for terminou sem interrupções.

Sentenças de Iteração

A sentença *for* em Python

- Para a maioria dos **laços de contagem** em Python, utiliza-se a função **range()**.
- Essa função gera uma **sequência de números inteiros** e é usada em laços for do tipo:

for variável in range(início, fim, passo):

corpo do laço

Sentenças de Iteração

A sentença *for* em Python

- Pode receber **um, dois ou três parâmetros**:
  - range(fim) → [0, 1, 2, 3, ..., fim-1]
  - range(início, fim) → [início, início+1, ..., fim-1]
  - range(início, fim, passo) → [início, início+passo, ...]
- Exemplos:
  - range(5) # [0, 1, 2, 3, 4]
  - range(2, 7) # [2, 3, 4, 5, 6]
  - range(0, 8, 2) # [0, 2, 4, 6]
- O valor final (**fim**) **nunca é incluído** na sequência.

Sentenças de Iteração

Laços controlados por contador em linguagens funcionais

- Nas **linguagens imperativas**, laços de contagem usam uma **variável contadora**.
- Já nas **linguagens funcionais puras**, **não existem variáveis mutáveis**.
- Em vez disso, a **repetição é controlada por recursão**:
  - Uma **função recursiva** substitui o laço.
  - Essa função recebe como parâmetros:
    - a **função corpo do laço** (ação a repetir)
    - e o **número de repetições**.

Sentenças de Iteração

Laços controlados por contador em linguagens funcionais

Exemplo em F#

```
let rec forLoop loopBody reps =
    if reps <= 0  then
        ()
    else
        loopBody()
        forLoop loopBody, (reps - 1);;
```

- *loopBody*: função com o corpo do laço
- *reps*: número de repetições
- *rec*: indica que a função é **recursiva**
- *()*: representa **nenhum valor** (semelhante ao void em C)
- Em F#, **todo if deve ter um else**, mesmo que vazio.

Sentenças de Iteração

Laços controlados logicamante

- O **controle da repetição** pode ser feito por **uma expressão booleana** em vez de um contador numérico.
- Para essas situações, um **laço controlado logicamente** é mais conveniente.
- Na verdade, **laços lógicos são mais gerais** do que os laços controlados por contador:
  - Todo laço de contagem pode ser construído a partir de um laço lógico.
  - O inverso não é verdadeiro.
- **Seleção e laços lógicos** são essenciais para expressar o controle de qualquer diagrama de fluxo.

Sentenças de Iteração

Questões de Projeto (Laços Lógicos)

- **O controle deve ser de pré-teste ou pós-teste?**
  - Define se a condição é testada **antes** de entrar no corpo do laço (como while) ou **depois** (como do...while).
- **O laço controlado logicamente deve ser uma forma especial de laço de contagem ou uma sentença separada?**
  - Decide se a linguagem trata o laço lógico como um **caso particular do laço for** (com contador implícito) ou como uma **construção própria**, distinta.

Sentenças de Iteração

Laços controlados logicamante

- As linguagens baseadas em C incluem laços controlados logicamente com:
  - **Pré-teste** → while (expressão_de_controle)
  - **Pós-teste** → do { ... } while (expressão_de_controle);
- A estrutura **while** executa o corpo do laço **enquanto** a condição for verdadeira.
- A estrutura **do...while** executa o corpo do laço **pelo menos uma vez** e repete enquanto a condição for verdadeira.

Sentenças de Iteração

Laços controlados logicamante

- Exemplo em C#:
  - No laço while, o teste é feito antes da execução.
  - No do-while, o corpo é executado antes da verificação.

```
sum = 0;
indat = Int32.Parse(Console.ReadLine());
while (indat >= 0) {
   sum += indat;
   indat = Int32.Parse(Console.ReadLine());
}
value = Int32.Parse(Console.ReadLine());
do {
    value /= 10;
    digits ++;
}  while (value > 0);
```

Sentenças de Iteração

Laços controlados logicamante

- Descrições semânticas (**while**):

```
while

loop:
   if expressão_de_controle is false goto out
   [corpo do laço]
   goto loop
out: . . .
```

Sentenças de Iteração

Laços controlados logicamante

- Descrições semânticas (**do-while**):

`do-while`

loop:
  [corpo do laço]
  `if` expressão_de_controle is true **`goto`** loop

Sentenças de Iteração

Laços controlados logicamante

- Exemplo Funcional em F#
- Laço lógico com **pré-teste**, simulado por recursão:
  - test é uma **função booleana** qu controla o laço.
  - body é a **função com o corpo** a ser repetido.
  - A recursão faz o papel do while, e o if controla a parada.

```
let rec whileLoop test body =
    if test() then
        body()
        whileLoop test body
    else
        ();;
```

Sentenças de Iteração

Mecanismos de controle de laços posicionados pelo usuário

- Em algumas situações, é conveniente que o programador escolha **onde o controle do laço** será verificado, não apenas no início ou no fim.
- Algumas linguagens permitem **controle de laço posicionado pelo usuário**, com mecanismos como **break**, **continue** e variantes rotuladas.
- Esses mecanismos oferecem **saídas ou saltos controlados**, sem recorrer a goto.

Sentenças de Iteração

Questões de projeto

- O mecanismo condicional deve ser parte integrante da saída?
- É possível sair apenas de um laço, ou também de laços externos que o envolvem?

Implementações em Linguagens

- Linguagens como **C, C++, Python, Ruby e C#** têm **saídas não rotuladas incondicionais** (break).
- **Java e Perl** também permitem **saídas incondicionais rotuladas.**

Sentenças de Iteração

- Exemplo Java: break rotulado

```
outerLoop:
  for (row = 0; row < numRows; row++)
    for  (col = 0; col < numCols; col++) {
      sum += mat[row][col];
      if (sum > 1000.0)
        break outerLoop;
    }
```

- O **rótulo outerLoop:** marca o laço externo.
- O comando **break outerLoop;** interrompe **diretamente o laço rotulado**, não apenas o laço interno.
- É útil quando há **laços aninhados**, e o programador precisa sair de mais de um nível de iteração.

Sentenças de Iteração

- Exemplo em C: **break** e **continue**

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) continue;
  sum += value;
}
```

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) break;
  sum += value;
}
```

- **continue**: pula o restante do corpo e volta ao topo do laço.
- **break**: termina completamente o laço.
- Ambas **evitam o uso explícito de goto**, mas devem ser usadas com moderação para preservar a legibilidade.

Sentenças de Iteração

Mecanismos de controle de laços posicionados pelo usuário

- break e continue atendem a uma **necessidade comum de desvios condicionais limitados**, substituindo o goto em muitos casos.
- Os destinos das saídas de laço **devem estar abaixo da saída** e **imediatamente após o bloco composto**.
- Apesar de úteis, **múltiplos breaks aninhados** podem prejudicar a legibilidade do código.

Sentenças de Iteração

Iteração baseada em estruturas de dados

- Uma **sentença de iteração geral** pode se basear em uma estrutura de dados e em uma função definida pelo usuário (**iterador**) para navegar pelos elementos da estrutura.
- O **iterador** é chamado no início de cada iteração e retorna um elemento da estrutura em uma ordem específica.
- Exemplo: uma função que percorre uma **árvore binária**, visitando cada nó exatamente uma vez.
  - A execução termina quando o iterador **não encontra mais elementos**.

Sentenças de Iteração

Iteração baseada em estruturas de dados

- Exemplo de iterador em C (árvore binária):

```
for (ptr = root; ptr == null; ptr = traverse(ptr)) {
   . . .
}
```

- Aqui, *traverse* é o **iterador**.
- Iteradores podem ser usados para percorrer listas, árvores, matrizes ou outros tipos de coleções.

Sentenças de Iteração

Iteração baseada em estruturas de dados

- Linguagens como **PHP** possuem funções como **current, next, reset** e **each**, que permitem percorrer coleções sequencialmente.

```
reset $list;
print ("First number: " + current($list) + "");
while ($current_value = next($list))
  print ("Next number: " + $current_value + "<br \>");
```

Sentenças de Iteração

Iteração baseada em estruturas de dados

- Com o avanço da **programação orientada a objetos**, iteradores passaram a ser fornecidos junto às **coleções de dados**.
- O Java introduziu a **sentença foreach** (versão 5.0) para simplificar iterações sobre objetos que implementam Iterable:
- Por exemplo, se tivéssemos uma coleção ArrayList de cadeias, chamada myList, a seguinte sentença iteraria por todos os seus elementos, configurando cada um como myElement:

```
for (String myElement : myList) { . . . }
```

Sentenças de Iteração

Iteração baseada em estruturas de dados

- Em **C# e F#**, há suporte similar via a interface IEnumerable e a sentença foreach:

```
List<String> names = new List<String>();
names.Add("Bob");
names.Add("Carol");
names.Add("Alice");
. . .
foreach (String name in names)
  Console.WriteLine(name);
```

Sentenças de Iteração

Iteração baseada em estruturas de dados

- Python itera diretamente sobre os elementos de uma coleção.
- Toda coleção em Python (listas, tuplas, conjuntos, dicionários, etc.) possui um **iterador interno** que o for usa automaticamente.

```
names = ["Bob", "Carol", "Alice"]

for name in names:
    print(name)
```

Desvio Incondicional

- A sentença de desvio incondicional (**goto**) transfere o controle de execução para uma posição especificada no programa.
- No fim dos anos 1960, discutia-se se esse tipo de desvio deveria fazer parte de linguagens de alto nível e, se sim, como restringi-lo.
- O **goto** é a sentença mais poderosa para controlar o fluxo de execução, mas também uma das mais perigosas.
- Seu uso sem restrições pode tornar programas difíceis de ler, pouco confiáveis e com alto custo de manutenção.

Desvio Incondicional

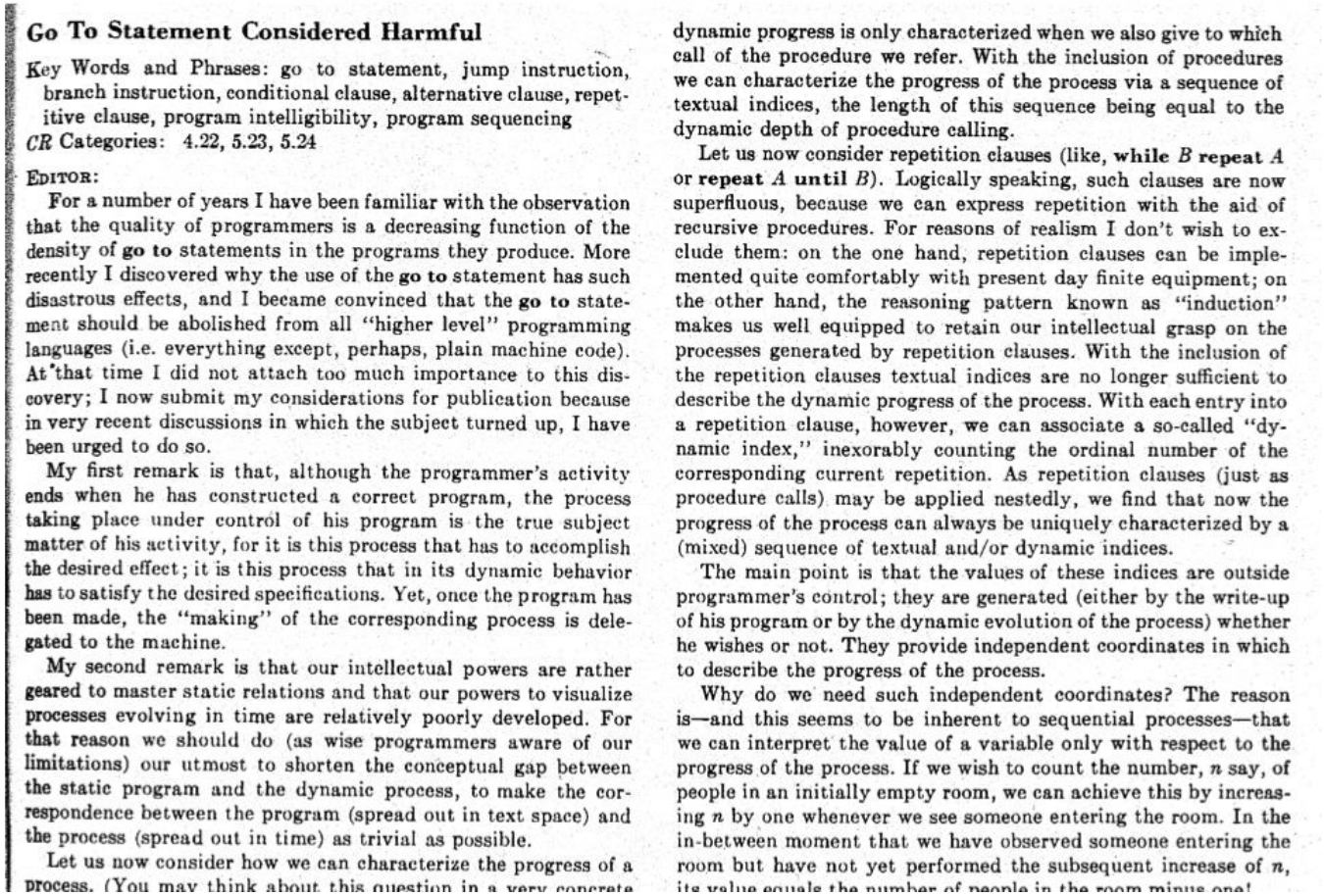
- O **goto** pode desviar o fluxo de execução para qualquer parte do código, ignorando a ordem textual.
- A legibilidade é melhor quando a execução segue uma ordem previsível, como de cima para baixo.
  - Por isso, restringir o uso do **goto** a desvios “para baixo” (como saídas de laços) reduz parte do problema.
- Ele pode ser útil em casos de **tratamento de erros** ou condições excepcionais, mas não deve substituir estruturas de controle adequadas.
- Linguagens como **Java, Python e Ruby** não possuem **goto**; enquanto **C e C++** o mantêm, e **C#** o inclui de forma restrita (como em switch).

Desvio Incondicional

"A sentença goto nos moldes de hoje é primitiva demais; é um convite muito enfático para bagunçar o programa de alguém."

(Dijkstra, 1968, Communications of the ACM)

Edgar Dijkstra: Go To Statement Considered Harmful



Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, *n* say, of people in an initially empty room, we can achieve this by increasing *n* by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of *n*, its value equals the number of people in the room minus one!

Desvio Incondicional

- **Edsger Dijkstra** publicou o primeiro texto amplamente conhecido sobre os perigos do **goto**.
- Seu artigo gerou grande debate: alguns pediram o banimento total, outros (como **Donald Knuth**) defenderam que há casos em que sua eficiência justifica o uso.
- A controvérsia levou à criação de práticas e padrões que limitam o **goto**, priorizando legibilidade e manutenção.

Desvio Incondicional

- As sentenças de saída de laço (como break e continue) podem ser vistas como **gotos restritos,** não prejudicam a legibilidade.
- Em alguns casos, **evitar completamente o goto** resulta em código mais complexo e menos legível; por isso, **uso criterioso e limitado** é aceitável.

Desvio Incondicional

- As sentenças de saída de laço (como break e continue) podem ser vistas como **gotos restritos,** não prejudicam a legibilidade.
- Em alguns casos, **evitar completamente o goto** resulta em código mais complexo e menos legível; por isso, **uso criterioso e limitado** é aceitável.

Paradigmas de Linguagens de Programação

Estruturas de controle no nível de sentença