Paradigmas de Linguagens de Programação

Prof. Sergio D. Zorzo
Departamento de Computação - UFSCar
1º semestre / 2013
Aula 6

Material adaptado do Prof. Daniel Lucrédio

Paradigma imperativo
Controle de fluxo

Controle de fluxo

- Algoritmo = lógica + controle
- Em programação lógica
  - Programador especifica lógica
  - Computador decide o controle
    - Regras de computação
    - Estratégia de busca
- Em programação funcional
  - Programador especifica as funções
  - Computador aplica as funções
    - Ordem pré-definida
- Em programação imperativa
  - Programador dita as regras (lógica e controle)

Controle de fluxo

- Três níveis
  - Nível de expressões
  - Entre as instruções
  - Entre blocos/unidades de programa

Controle de fluxo em nível de expressões

Expressões aritméticas

- Operadores unários / binários / ternários
- Como é feita a avaliação de uma expressão?
- Depende de várias questões de projeto
  - Regras de precedência
  - Regras de associatividade
  - Ordem de avaliação
  - Efeitos colaterais
  - Avaliação curto-circuito
- Normalmente o programador consegue controlar o fluxo em uma expressão
  - Ou pelo menos, ele deve conhecer as regras para a linguagem, para saber qual fluxo será tomado

Precedência/associatividade de operadores

- A maioria das linguagens segue as convenções matemáticas
- Alguns destaques
  - Na linguagem APL, todos os operadores tem o mesmo nível de precedência
    - A + B * C executa a adição primeiro
  - Exponenciação em ADA não é associativa
    - A ** B ** C é ilegal
- Operador pode modificar o fluxo com o uso de parênteses

Expressões condicionais

- Operador ternário
- Controle explícito de fluxo em uma única expressão

```
media = (cont == 0) ? 0 : soma / cont;
```

Ordem de avaliação de operandos

- Ordem não é importante
  - Ex: A * B, tanto faz avaliar A ou B primeiro
  - Claro: operandos dentro de parênteses devem ser avaliados antes de utilizar o resultado
- Porém, caso a avaliação de um operando tenha **efeitos colaterais**, a ordem importa

Efeitos colaterais

- Quando uma função modifica um de seus parâmetros ou uma variável global
- Ex:
- Suponha que fun(a) retorne a/2, e modifica o parâmetro a para 20 (efeito colateral)

```
a = 10
b = a + fun(a)
```

- Se a for avaliado primeiro, o resultado é 15
- Se fun(a) for avaliado primeiro, o resultado é 25

Efeitos colaterais

- Não podem ser eliminados
  - Reduziria a flexibilidade
- Solução: fixar a ordem de avaliação
  - Da esquerda para a direita

Avaliação curto-circuito

- Determinação do resultado de uma expressão sem avaliar todos os operandos/operadores
- Ex:

```
(13 * a) * (b / 13 - 1)
```

- Se a = 0, o valor independe de (b / 13 – 1)
  - Em expressões aritméticas, isso normalmente não é feito, por ser de difícil implementação
- Em expressões lógicas booleanas, é mais fácil detectar curto-circuitos
- Ex:

```
(a >= 0) and (b < 10)
```

Avaliação curto-circuito

- Ex: Java

```
indice = 0;
while((indice < complis) &
      (lista[indice] != chave))
  indice++;
```

- Se chave não estiver em lista, a rotina terminará com erro de índice de vetor fora do intervalo permitido
- Também é necessário considerar efeitos colaterais

Controle de fluxo em nível de instruções

Controle de fluxo em nível de instruções

- Comandos específicos para controle de fluxo
  - Desvio incondicional
  - Seleção bidirecional
  - Seleção múltipla
  - Iteração / laços

Desvio incondicional

- Instrução mais simples e poderosa de controle de fluxo
  - Altamente flexível
  - Todas as outras estruturas de controle podem ser construídas com GOTO e um seletor
    - De fato, era o mecanismo original das linguagens de máquina e de montagem
- Formato geral: GOTO rótulo/linha
- Bastante popular no início
  - Mas pode gerar problemas de legibilidade, confiabilidade e manutenibilidade
- Algumas linguagens eliminaram o GOTO
  - Ex: Java (apesar de ser palavra reservada)

Desvio incondicional

- A alternativa são as estruturas típicas de controle, que veremos a seguir
  - Promovem encapsulamento sintático
  - Ou seja, a lógica de controle fica mais evidente no programa
- Mas existem casos onde o GOTO é mais conveniente

Desvio incondicional

- Ex: encontrar a primeira linha de uma matriz de números inteiros n x n que só possui zeros

```
for i := 1 to n do
  begin
  for j := i to n do
    if x[i, j] <> 0
      then goto rejeita;
    writeln('Primeira linha só com zeros é', i);
    break;
rejeita:
  end;
```

Desvio incondicional

- Na prática, a grande maioria dos usos convenientes do GOTO pode ser feita através de outras instruções equivalentes
  - Veremos mais adiante: exit, break, continue
- Mas o GOTO permanece em muitas linguagens atuais
  - A regra não deve ser: evite GOTO
  - Mas sim: evite GOTO quando existir alternativa mais elegante

Instruções de seleção

- No início, não haviam instruções compostas
- Ex: FORTRAN IV

```
      IF(FLAG .NE. 1) GO TO 20
      I = 1
      J = 2
20    CONTINUE
```

- Era necessário o uso do GO TO
  - Lógica negativa muitas vezes

Instruções de seleção bidirecional

- Primeiro seletor bidirecional: ALGOL 60
  - Cláusulas then e else
  - Instruções compostas
  - Modelo utilizado até hoje

```
if(expressão_booleana) then
    instrução
else
    instrução
```

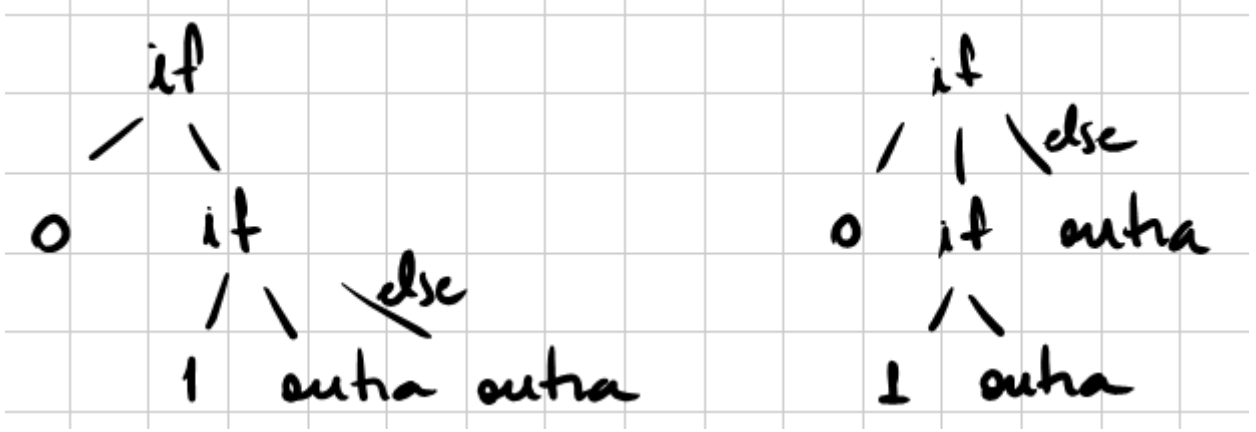
Seletores aninhados

- Considere a gramática

```
declaração → if-decl | outra
if-decl → if (exp) declaração | if (exp)
  declaração else declaração
exp → 0 | 1
```

- Verifique que há duas árvores de análise sintática para a seguinte cadeia

```
if(0) if (1) outra else outra
```



Seletores aninhados

- Na maioria das implementações de compiladores, há um conflito empilha-reduz
  - Solução adotada na maioria das linguagens modernas: na dúvida, empilhe
  - Resultado obtido: o **else** sempre faz par com a cláusula **then** não-emparelhada mais recentemente.
- Ex Java:

```
if(soma == 0)
   if(cont == 0)
     resultado = 0;
   else
     resultado = 1;
```

Seletores aninhados

- Outras soluções
  - ALGOL 60: não é permitido aninhar seletores diretamente. É necessário criar uma instrução composta (begin-end)

```
if soma = 0 then
   begin
   if cont = 0 then
      resultado := 0
   else
      resultado := 1
   end
```

Seletores aninhados

- Outras soluções
  - PERL: todas as cláusulas then e else devem ser compostas
  - ADA: palavras especiais de fechamento de seleção

```
if A > B then
   SOMA := SOMA + A;
   ACONT := ACONT + 1;
else
   SOMA := SOMA + B;
   BCONT := BCONT + 1;
end if;
```

Construções de seleção múltipla

- FORTRAN I
  - Versão rudimentar, centrada no GO TO

```
   GO TO (10, 20, 30), expressão_inteira
10 ...
   ...
   GO TO 40
20 ...
   ...
   GO TO 40
30 ...
   ...
40 ...
```

Construções de seleção múltipla

- Seletores modernos
  - Encapsulamento sintático
  - Maior legibilidade
  - Sem necessidade de controle explícito de saltos
- Ex: ALGOL-W, PASCAL

```
case indice of
  1, 3: begin
        impar := impar + 1;
        somaimpar := somaimpar + indice
        end;
  2, 4: begin
        par := par + 1;
        somapar := somapar + indice
        end
  else writeln('Erro na instrução case, indice = ', indice)
end
```

Construções de seleção múltipla

- JAVA, C, C++

```
switch (indice) {
  case 1:
  case 3: impar += 1;
          somaimpar += indice;
          break;
  case 2:
  case 4: par += 1;
          somapar += indice;
          break;
  default: printf("Erro na switch, indice = %d\n", indice);
}
```

Instruções iterativas

- Execução de uma instrução ou um conjunto de instruções
  - zero, uma ou mais vezes
- Essência do poder computacional da arquitetura de von Neumann
  - Contraste com a recursividade
  - Recurso matemático (utilizado também nos paradigmas lógico e funcional)
- Quatro tipos
  - Laços controlados por contador
  - Laços controlados logicamente
  - Laços baseados em estruturas de dados
  - Controle definido pelo programador

Laços controlados por contador

- Variável de laço
- Valores de início e fim
- Tamanho do passo
- FORTRAN I

```
DO rótulo variável = inicial, final [, tamanho do passo]
```

- ALGOL 60

```
for indice := 1, 4, 13, 41,
              step 2 until 47,
              3 * indice while indice < 1000,
              34, 2, -24 do
  soma := soma + indice
```

Ex: soma os valores 1, 4, 13, 41, 43, 45, 47, 147, 441, 34, 2, -24

- PASCAL

```
for var := val_inicial (to | downto) val_final do instrução
```

Laços genéricos

- C / C++ / JAVA / C#

```
for(expressão_1; expressão_2; expressão_3)
    corpo do laço
```

- Todas as expressões são opcionais, assim como o corpo do laço
- Equivale semanticamente a

```
        expressão_1
laço:
        if expressão_2 == false goto fora
        [corpo do laço]
        expressão_3
        goto laço
fora: ...
```

Laços controlados logicamente

- Instruções simples
- Questão principal: pré-teste ou pós-teste
- Ex: C++

```
while (expressão)
   corpo do laço

do
   corpo do laço
while (expressão)
```

Laços baseados em estruturas de dados

- macro DOLIST em LISP
- A idéia é percorrer uma estrutura (lista, matriz, etc)
- Ex: Perl

```
@nomes = ("Bob", "Carol", "Ted", "Beelzebub");
...
foreach $nome (@nomes) {
  print $nome;
}
```

- Ex: Java

```
animais = new Animal[] { toto, saddam, mimi, nemo,
  alazao };
for (Animal a:animais) {
  a.examinar();
}
```

Controle definido pelo programador

- Ex: laço infinito em Ada

```
loop
  ...
end loop
```

- O programador precisa intervir

```
exit [rótulo do laço] [when condição]
```

Controle definido pelo programador

- Ex Ada:

```
LACO_EXTERNO:
   loop
LACO_INTERNO:
     loop
       SOMA := SOMA + MAX(LINHA, COL);
       exit LACO_EXTERNO when SOMA > 1000.0;
     end loop LACO_INTERNO;
   end loop LACO_EXTERNO;
```

Controle definido pelo programador

- Ex Java (C, C++ e C# são similares)

```
int soma = 0; int linha = 0; int coluna = 0;
laco_externo:
  for(;;) {
    linha ++;
    laco_interno:
      for(;;) {
        coluna ++;
        soma += Math.max(linha, coluna);
        if(soma > 1000) break laco_externo;
      }
  }
System.out.println(soma);
```

Controle definido pelo programador

- C, C++, C#, Java

```
continue [rótulo]
```

  - Transfere o fluxo para o mecanismo de controle do laço especificado

```
laco:
while (soma < 1000) {
   getnext(valor);
   if (valor < 0) continue laco;
   soma += valor;
}
```

Controle definido pelo programador

- C, C++, C#, Java
- Tanto break quanto continue podem ser usados sem rótulos
  - Será considerado a estrutura envolvente mais próxima
  - Ex:

```
while (soma < 1000) {
   getnext(valor);
   if (valor < 0) break;
   soma += valor;
}
```

Controle de fluxo em nível de unidade

Controle de fluxo em nível de unidade

- Mecanismos que permitem fazer chamadas de unidades
  - Chamadas de unidades **explícitas**: funções, procedimentos (subprogramas).
  - Chamadas de unidades **implícitas**: tratadores de exceção, corrotinas, unidades concorrentes.

Subprogramas

- Essência da programação estruturada
- Também chamada de programação procedural
  - Baseada em procedimentos
- Conceito de subprogramas
- Função – abstrai uma expressão a ser avaliada
  - Funções retornam valores.
  - Exemplo: cálculo de fatorial de um dado número:
  - fatorial(n) deve retornar n!
  - Efeito colateral: quando os parâmetros da função retornam valores.
- Procedimento – abstrai um comando a ser executado
  - Modifica variáveis
  - Exemplo: ordenação de um vetor de números.
  - ordena(v) deve ordenar o vetor v.

Subprogramas

- Ex PASCAL:

```
function <nome da função> ( <lista de argumentos> ) : <tipo>;
  <corpo da função>
end
procedure <nome> (<parâmetros>);
  <corpo>
end
```

- Ex C:

```
<tipo do resultado> <nome> ( <declaração de parâmetros formais>
  )
{
  <lista de declarações>
  <lista de comandos>
}
```

Subprogramas

- Conceito de protótipo de um subprograma
  - Aquilo que é necessário saber para utilizá-lo
  - Nome, parâmetros e tipo de retorno (se houver)
  - Exceções lançadas (em Java)
- C++ separa a declaração (protótipo) da definição
- Veremos mais detalhes sobre como funciona a chamada a subprogramas posteriormente

Parâmetros

- Há duas formas pelas quais um subprograma pode acessar dados
  - Acesso direto a variáveis não-locais
    - Declaradas em outro local, mas visíveis
  - Passagem de parâmetros
    - Computação parametrizada
- Parâmetros são uma forma mais conveniente
  - Nomes locais
  - Maior modularização
  - Verificação de tipos
- É inclusive possível passar subprogramas como parâmetros

Parâmetros

- **Parâmetro formal** – identificadores usados no cabeçalho do subprograma (definição do subprograma)
- **Parâmetro real** – identificadores, expressões ou valores usados na chamada do subprograma.
- **Argumento** – usado como sinônimo de parâmetro real ou para referir o valor passado do parâmetro real para o formal.

Parâmetros

- A maioria das linguagens usa um **critério posicional para** amarração de argumentos e parâmetros
- Definição do procedimento – pi são parâmetros:

```
procedure S( p1; p2; ....; pn);
..
end;
```

- Chamada do procedimento – ai são argumentos:

```
S( a1; a2; .....; an);
```

pi corresponde a ai para i=1,...n.

Parâmetros

- Em Ada pode-se usar palavras-chave:

```
SOMADOR(COMPRIMENTO => COMP, LISTA => L)
```

- Desvantagem de precisar conhecer os nomes dos parâmetros formais

Parâmetros

- Conceituamente, existem três modelos semânticos
  - Parâmetros de entrada
  - Parâmetros de saída
  - Parâmetros de entrada/saída
- Para implementar esses modelos, existem diversas opções
  - Passagem por cópia
  - Passagem por referência
  - Passagem de nome

Parâmetros

- Passagem por cópia
  - Valores são copiados na chamada/saída de um subprograma
  - Serve para os três modelos semânticos
- Passagem por valor
  - Valores são copiados para novas variáveis, locais ao subprograma
  - Modificar o valor dos parâmetros formais não causa impacto nos parâmetros reais
    - Implementa o modelo de parâmetro de entrada

Parâmetros

- Passagem por resultado
  - Parâmetros formais são copiados para parâmetros reais, imediatamente antes do retorno da chamada do subprograma
  - Especificar parâmetros reais não tem sentido/utilidade
    - É necessário especificar variáveis, já que expressões ou literais não podem ser modificadas
  - Modificar o valor dos parâmetros formais causa impacto nos parâmetros reais
    - Implementa o modelo de parâmetro de saída

Parâmetros

- Passagem por valor-resultado
  - Combinação das abordagens anteriores
  - Parâmetro real é copiado para o parâmetro formal na entrada
  - Parâmetro formal é copiado para o parâmetro real na saída
    - Modelo entrada/saída

Parâmetros

- Passagem por cópia
  - Custo extra da cópia
  - Memória e processamento adicionais para efetivamente duplicar os valores
  - Em casos extremos (ex: matrizes muito grandes) pode não ser uma boa opção

Parâmetros

- Passagem por referência
- Unidade chamadora passa para a unidade chamada o **endereço** do parâmetro real
- A variável usada como parâmetro real é compartilhada e pode ser modificada
- Parâmetro formal é um apelido para o parâmetro real
  - Modelo entrada/saída

Parâmetros

- Passagem por referência
  - Maior economia de recursos
  - Não há a necessidade de cópia
- Problemas:
  - Necessidade de cautela quando usado para o modelo de entrada
    - Ex: const em C / C++ para prevenir alterações
  - Possibilita a criação de apelidos
    - Duas variáveis que na verdade são a mesma
    - Baixa legibilidade / confiabilidade
    - Ex: fun(&lista[i], &lista[j]);
    - Se i e j forem iguais, os parâmetros formais serão apelidos

Parâmetros

- Passagem de nome
  - A amarração do parâmetro à posição não é feita na hora da chamada, mas a cada vez que ele é usado na unidade chamada
  - Modelo entrada/saída
  - Vinculação/amarração tardia
- Vantagens: flexibilidade
- Desvantagens:
  - custo da vinculação tardia
  - complexidade no uso pode prejudicar legibilidade e capacidade de escrita

Parâmetros

- Ex – passagem de nome:

```
procedure BIGSUB;
  integer GLOBAL;
  integer array LIST [1:2];
  procedure SUB (PARAM);
    integer PARAM;
    begin
      PARAM := 3;
      GLOBAL := GLOBAL + 1;
      PARAM := 5
    end;
  begin
    LIST[1] := 2;
    LIST[2] := 2;
    GLOBAL := 1;
    SUB(LIST [GLOBAL])
  end;
```

Ao final dessa execução, LIST tem valores 3 e 5, atribuídos em SUB.

Exemplo – C

```
void troca1 (int a, int b) {
   int temp = a;
   a = b;
   b = temp;
}
...
troca1(c,d)
```

Resultado – C

```
a = c
b = d
temp = a
a = b
b = temp
```

- Ou seja c e d não são modificados
  - Passagem por valor

Exemplo – Pascal

```
procedure troca1 (a, b: integer) {
  temp: integer;
  begin
  temp := a;
  a := b;
  b := temp;
  end;
```

Resultado – Pascal

```
a = c
b = d
temp = a
a = b
b = temp
```

- Ou seja c e d não são modificados
  - Passagem por valor

Exemplo – C

```
void troca2 (int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
...
troca2(&c,&d)
```

Resultado – C

```
a = &c
b = &d
temp = *a
*a = *b
*b = temp
```

- Ou seja c e d são modificados
  - Passagem por referência
  - Não seria possível em Java

Exemplo – Pascal

```
procedure troca2 (var a, b: integer) {
  temp: integer;
  begin
  temp := a;
  a := b;
  b := temp;
  end;
```

Exemplo - Ada

```
procedure troca3(a : in out integer,
                 b : in out integer) is
  temp : integer;
  begin
  temp := a;
  a := b;
  b := tempo;
  end troca3;
...
troca3(c, d);
```

Resultado - Ada

```
end_c = &c
end_d = &d
a = *end_c
b = *end_d
temp = a
a = b
b = temp
*end_c = a
*end_d = b
```

- Ou seja c e d são modificados
  - Mas em uma passagem por valor-resultado
  - Idêntica à passagem por referência
    - Desde que não haja a criação de apelidos

Resultado - Ada

```
Chamada: troca3(i, lista[i]);
Resultado:
end_i = &i
end_listai = &lista[i]
a = *end_i
b = *end_listai
temp = a
a = b
b = temp
*end_i = a
*end_listai = b
```

- Ou seja i e lista[i] são modificados corretamente
  - Porque os endereços são computados na entrada
  - Se os endereços fossem computados no momento de retorno, o resultado seria diferente

Exemplo

```
int i = 3;
void fun(int a, int b) {
  i = b;
}
void principal() {
  int lista[10];
  lista[i] = 5;
  fun(i, lista[i]);
}
```

Resultado: considerando passagem por valor-resultado

```
end_i = &i
end_listai = &lista[i]
a = *end_i
b = *end_listai
i = b
*end_i = a
*end_listai = b
```

- Resultado final: i vale 3

Resultado: considerando passagem por referência

```
a = &i
b = &lista[i]
*i = *b
```

- Resultado final: i vale 5
  - Se o procedimento alterasse b (ex: para 20), seria alterado o valor de lista[3] para 20, e não lista[5]
  - Já que o mesmo é calculado no início do procedimento

Passagem de nomes de subprogramas como argumentos

- Exemplo: um programa que precisa avaliar uma função matemática
  - A função pode ser qualquer uma
  - Solução elegante: passar a função como argumento, e avaliá-la dentro de um subprograma
- Vimos algo semelhante em LISP
  - É uma linguagem funcional, portanto possui esse suporte
- A maneira com que a chamada de subprogramas é implementada influencia nessa característica
  - Veremos mais sobre isso na próxima aula

Resumo

Resumo

- Vimos estruturas de controle
  - Em nível de expressões
    - Algum controle do programador
    - Mas na maioria dos casos, por convenção
  - Em nível de instruções
    - Mais usadas
    - Seleção / iteração
  - Em nível de unidades
    - Explícitas (chamadas a subprogramas)
    - Implícitas (não vimos em detalhes)
- Subprogramas
  - Principal nesse nível são os parâmetros e as diferentes possibilidades de comunicação de dados entre chamador e chamado

Resumo

- Na próxima aula veremos como isso pode ser implementado
- Ajuda a entender melhor o funcionamento dos programas
  - E a essência do paradigma imperativo

Fim