Capítulo V – Expressões e Comandos

"Computadores são bons para seguir instruções, mas não para ler sua mente."
Donald Knuth

Esse capítulo aborda os conceitos envolvidos na construção e execução de expressões e comandos nas LPs. São apresentados e discutidos os tipos fundamentais de expressões e comandos. Especial atenção é dada aos estudos sobre os modos de avaliação de expressões e sobre a presença de efeitos colaterais em expressões e comandos.

5.1 Expressões

Uma expressão é uma frase do programa que necessita ser avaliada e produz como resultado um valor. Expressões são caracterizadas pelo uso de operadores, pelos tipos dos operandos e pelo tipo de resultado que produzem. Operadores designam o tipo de operação a ser realizada. Operandos indicam os valores sobre os quais a operação deve ser avaliada. O resultado é o valor produzido pela avaliação da expressão.

Expressões podem ser simples ou compostas. Expressões simples são aquelas que envolvem apenas um operador. Expressões compostas envolvem mais de um operador.

Um operador possui uma aridade (isto é, o número de operandos requeridos para a realização da operação): operadores unários possuem aridade um (requerem um operando); operadores binários possuem aridade dois (requerem dois operandos); operadores ternários possuem aridade três (requerem três operandos); e assim progressivamente. Operadores eneários possuem aridade variável (podem ter qualquer número de operandos).

Operadores podem ser pré-existentes na LP ou podem ser criados pelos programadores através da definição de funções. Normalmente, a maior parte dos operadores pré-existentes de uma LP são unários e binários (com predominância dos binários). A presença de operadores pré-existentes de maior aridade é rara. Por exemplo, C, C++ e JAVA oferecem um único operador ternário e nenhum de aridade superior.

Definições de funções permitem aos programadores criar operadores de qualquer aridade (inclusive os de aridade mais alta). Para isso, basta definir a lista de parâmetros da função com o número correspondente a aridade desejada. Contudo, não é muito conveniente criar operadores de arida-

de muito elevada pois isso torna o programa menos legível, além de reduzir a sua redigibilidade, desestimulando o uso desses operadores.

Operadores eneários são raros nas LPs mais conhecidas. JAVA e PASCAL não possuem qualquer operador eneário. C e C++ permitem a construção de funções com número de parâmetros variável, possibilitando assim aos programadores construírem operadores eneários. Contudo, não existe qualquer operador eneário pré-existente nessas linguagens. APL e dialetos de LISP oferecem operadores eneários.

Algumas LPs possuem um operador de composição para permitir que operadores sejam operandos de outros operadores. Dessa forma, o programador pode criar novos operadores a partir da combinação dos operadores existentes. A função *impar* do exemplo 2.19 ilustra esse tipo de operação em ML. APL também possibilita operações de composição de operadores.

Operadores podem ser usados com notação prefixada (o operador é colocado antes dos operandos), infixada (o operador é colocado entre os operandos) e posfixada (o operador é colocado após os operandos). Normalmente, operadores binários são usados na forma infixada e os demais operadores na forma prefixada. Uma mesma linguagem pode adotar diferentes tipos de notação para os seus diversos operadores. Por exemplo, C e C++ usam a notação prefixada para a maioria dos seus operadores unários (tais como, *! & sizeof*), a notação infixada para seus operadores binários e ternários (tais como, ** / % && ?:*) e a notação posfixada para alguns de seus operadores unários (++ --). Além disso, algumas LPs permitem o uso de mais de uma notação para um mesmo operador. Dialetos de LISP, por exemplo, permitem que se use o operador + com as notações prefixada e infixada.

Expressões podem realizar operações de diferentes naturezas. Algumas produzem valores. Algumas consultam ou analisam propriedades de uma variável, constante ou tipo. Outras modificam os valores de variáveis. Por fim, outras destroem variáveis.

5.1.1 Tipos de Expressões

Expressões podem ser classificadas de acordo com a natureza da operação que realizam, com os tipos de seus operandos e com o tipo do resultado que produzem.

5.1.1.1 Literais

Expressões literais são consideradas como o tipo mais simples de expressão. Essa expressão é usada para produzir um valor fixo e manifesto de

um tipo e não demanda o uso de um operador explícito. A linha seguinte ilustra vários exemplos de expressões literais em C:

2.72 99 0143 'c' 0x63

A expressão *2.72* produz um valor do tipo real. Todas as demais expressões da linha produzem o valor inteiro *99*.

5.1.1.2 Agregação

Uma expressão de agregação é utilizada para construir um valor composto a partir de seus componentes. Os valores dos componentes do agregado resultante são determinados através da avaliação das subexpressões que compõem a expressão de agregação. Um operador de agregação é normalmente denotado por marcadores de início e fim da expressão de agregação. Geralmente, as diferentes subexpressões que compõem a expressão são separadas por um símbolo separador. O exemplo 5.1 mostra vários exemplos de expressões de agregação em C.

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

Exemplo 5. 1 - Expressões de Agregação em C

Observe que, nas quatro primeiras linhas do exemplo 5.1, o operador de agregação é denotado pelos caracteres de abre (*{*) e fecha (*}*) chaves. Nas três primeiras linhas as subexpressões são separadas pelo caractere vírgula. Note que o tamanho do vetor *c* é definido pelo número de subexpressões no agregado. Na quarta linha, uma única subexpressão é utilizada para iniciar todos os valores do vetor *b*. Por isso, não se usa qualquer caractere de separação.

A expressão de agregação da última linha do exemplo 5.1 é equivalente a expressão de agregação da terceira linha. Em razão de agregados de caracteres serem muito utilizados, C fornece uma comodidade sintática para a sua utilização. Nessa notação simplificada (tal como a da última linha do exemplo) o operador de agregação é denotado pelos caracteres aspas duplas e não é preciso usar caracteres de separação.

As subexpressões utilizadas para formar o agregado podem ser estáticas (avaliadas em tempo de compilação) ou dinâmicas (avaliadas em tempo de execução). O exemplo 5.2 ilustra o uso de expressões de agregação para a inicialização de três vetores em C.

```
void f(int i) {
  int a[] = {3 + 5, 2, 16/4};
```

```
  int b[] = {3*i, 4*i, 5*i};
  int c[] = {i + 2, 3 + 4, 2*i};
}
```

Exemplo 5. 2 - Expressões de Agregação Estáticas e Dinâmicas

Observe que o agregado do primeiro vetor do exemplo 5.2 só contém subexpressões estáticas. O agregado do segundo vetor só contém subexpressões dinâmicas. Já o agregado do terceiro vetor contém subexpressões estáticas e dinâmicas.

C apenas possibilita o uso de expressões de agregação em operações de inicialização e para a criação de cadeias de caracteres constantes. Assim, não é possível usar uma expressão de agregação para atribuir um valor a uma estrutura ou a um vetor. ADA não possui essa restrição, permitindo o uso de agregados tanto em inicializações como em outras operações, tal como a de atribuição. O exemplo 5.3 ilustra o uso de agregados em ADA para a inicialização e atribuição de registros.

```
type data is record
        dia : integer range 1..31;
        mes : integer range 1..12;
        ano : integer range 1900..2100;
end record;
aniversario: data;
data_admissao: data:= (29, 9, 1989);
aniversario :=(28, 1, 2001);
data_admissao  := (dia => 5, ano => 1980, mes => 2);
```

Exemplo 5. 3 - Expressões de Agregação em ADA

Note que as três últimas linhas do exemplo 5.3 utilizam agregados para inicializar *data_admissao* e para atribuir valores a *aniversario* e *data_admissao*. Observe ainda que existem duas maneiras de criar agregados de registros. Uma utiliza a correspondência posicional entre os valores do agregado e os campos do registro. A outra utiliza correspondência nominal entre os valores e os campos, não sendo necessário colocar os valores na mesma ordem em que foram listados na definição do tipo.

5.1.1.3 Expressões Aritméticas

Expressões aritméticas são expressões similares às existentes nessa área da matemática. Elas envolvem operandos e produzem valores de tipos numéricos (inteiros, decimais, ou ponto flutuante). Exemplos de expressões aritméticas são aquelas que denotam as operações de soma, subtração, multiplicação e divisão. C possui os seguintes operadores aritméticos: +, –, *, / e %.

Os operadores + e – podem ser usados como operadores unários ou binários. Quando usados como binários, eles denotam respectivamente as operações de soma e subtração. O operador unário – provoca a inversão da sinalização do valor do operando, isto é, um valor positivo se transforma no mesmo valor negativo e vice-versa. O operador unário + equivale a uma função identidade, retornando o mesmo valor do operando sem qualquer alteração.

Os operadores binários *, % e / denotam respectivamente as operações de multiplicação, resto da divisão inteira e divisão (inteira e real). O operador / em expressões aritméticas pode denotar a operação de divisão inteira (quando os operandos são de tipos inteiros) ou a operação de divisão real (quando pelo menos um dos operandos é real). O exemplo 5.4 mostra o uso do operador / denotando divisão inteira e divisão real.

```
float f;
int num = 9;
f = num/6;
f = num/6.0;
```

Exemplo 5. 4 - Divisão Inteira e Divisão Real em C

No exemplo 5.4, a variável *f* recebe o valor *1.0* na primeira atribuição e o valor *1.5* na segunda atribuição.

5.1.1.4 Expressões Relacionais

Uma expressão relacional é usada para comparar os valores de seus operandos. Os valores dos operandos devem ser de um mesmo tipo. Tipicamente, o resultado de uma expressão relacional é um valor booleano. Contudo, em LPs que não possuem o tipo booleano (tal como C), o resultado de uma expressão relacional é um valor que indica se a expressão relacional é verdadeira ou falsa.

C possui os seguintes operadores relacionais: ==, *!*=, >, <, >= e <=. Os operadores maior > , menor < , maior ou igual >= e menor ou igual <= devem ser aplicados sobre operandos de um mesmo tipo cujos valores sejam enumeráveis. Os operadores de igualdade == e desigualdade *!*= não necessitam cumprir essa exigência.

5.1.1.5 Expressões Booleanas

Expressões booleanas realizam operações da Álgebra de Boole. Os operandos das expressões booleanas são valores de tipo booleano ou equivalente (quando a LP não oferece esse tipo). O tipo do resultado de uma expressão booleana também é um valor booleano ou equivalente.

C possui o operador booleano unário de negação *!* (conhecido como **não lógico**) e os operadores booleanos binários de conjunção *&&* e disjunção || (mais conhecidos respectivamente como **e** e **ou lógicos**). Além desses, JAVA possui ainda os operadores *&* e | que realizam as mesmas operações de *&&* e ||. A necessidade para a existência desses dois operadores adicionais para realizar as operações de **e** e **ou lógico** está relacionada com a forma como essas operações são implementadas computacionalmente (esse tópico será melhor compreendido na seção 5.1.2.4).

5.1.1.6 Expressões Binárias

Algumas LPs fornecem operadores para a construção de expressões binárias. Essas expressões realizam operações sobre conjuntos de dígitos binários, isto é, manipulam bits. A vantagem de se oferecer expressões binárias em LPs é permitir a criação de programas com tempo de processamento mais eficiente e que também consomem menos memória.

C oferece seis operadores para a construção de expressões binárias: *&*, |, ^, <<, >> e ~. Eles podem ser aplicados a operandos de tipos inteiros. O operador *&* realiza a operação de conjunção binária (AND). O operador | realiza a operação de disjunção inclusiva (OR). O operador ^ realiza a operação de disjunção exclusiva (XOR). Os operadores << e >> realizam respectivamente as operações de deslocamento à esquerda (LEFT SHIFT) e deslocamento à direita (RIGHT SHIFT). Todos esses operadores são binários. O operador unário ~ realiza a operação de complemento a um (NOT). O exemplo 5.5 mostra o uso de cada um desses operadores em C.

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

Exemplo 5. 5 - Expressões Binárias em C

Observe que os operandos dos operadores binários são de tipos inteiros diferentes no exemplo 5.5. Nesses casos, ocorre uma conversão implícita do valor do tipo de menor intervalo para um valor equivalente no tipo de maior intervalo.

5.1.1.7 Expressões Condicionais

Expressões condicionais são compostas por várias (pelo menos, mais que uma) subexpressões e retornam como resultado o valor de exatamente uma dessas subexpressões. São usadas para avaliar expressões condicionalmente de acordo com o contexto. O exemplo 5.6 mostra o uso de duas expressões condicionais em ML:

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

Exemplo 5. 6 - Expressões Condicionais em ML

A primeira linha do exemplo 5.6 mostra uma expressão condicional *if* formada por três subexpressões. Caso a subexpressão *a > b* seja verdadeira, o valor retornado pela expressão condicional será o resultado da subexpressão *a –3*. Caso contrário, o valor retornado será o resultado da subexpressão *b + 5*. Já a expressão condicional *case*, mostrada a partir da segunda linha do exemplo 5.6, possui cinco subexpressões. Se o valor da subexpressão *i* for *1*, o resultado da expressão condicional corresponderá ao resultado da expressão condicional *if j > 5 then j - 8 else j + 5*. Se o valor de *i* for 2, o resultado será *2*j*. Se o valor for *3*, o resultado será *3*j*. Por fim, qualquer outro valor de *i* fará a expressão condicional retornar o valor de *j*.

C, C++ e JAVA fazem uso do operador ternário *?:* para oferecer um único tipo de expressão condicional. O exemplo 5.7 mostra dois exemplos do uso de expressões condicionais em JAVA. Na primeira linha, a expressão condicional retornará o maior valor entre *x* e *y*. Esse valor será atribuído a variável *max*. Na segunda linha, a expressão condicional é usada para avaliar se *z* é par ou não.

```
max = x > y ? x : y;
par  = z % 2 == 0 ? true : false;
```

Exemplo 5. 7 - Expressões Condicionais em JAVA

Algumas LPs (tais como, PASCAL e ADA) não oferecem expressões condicionais, forçando os programadores a utilizarem os comandos condicionais. O seguinte comando condicional é usado para encontrar o maior valor entre os números *x* e *y* em ADA.

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

Observe que o comando obriga uma repetição da atribuição à variável *max*, tornando-se assim menos redigível.

5.1.1.8 Chamadas de Funções

Chamadas de funções também são expressões. Elas produzem um resultado através da aplicação de um operador (o nome da função) a um ou mais operandos (os valores correspondentes aos parâmetros da função). Por exemplo, na chamada de função *f(a)*, *f* é o operador e *a* é o operando. O resultado retornado por *f(a)* é o resultado dessa expressão.

Funções possibilitam a criação de expressões com qualquer aridade. Por exemplo, se quisermos definir uma expressão de aridade **m**, basta definir uma função com **m** parâmetros. Nesse caso, o formato mais comum da chamada é o seguinte:

$f(a_1, a_2, ..., a_m)$

Esse formato de chamada de função é conhecido como posicional e é adotado pela maior parte das LPs (por exemplo, C, C++ e JAVA). Existe um outro formato de chamada de função por palavras chave (usado em ADA, por exemplo). A discussão sobre esses formatos será aprofundada no capítulo 6.

Na maioria das LPs, a única forma de se chamar uma função é através do uso exclusivo de seu identificador. Por exemplo, ao se designar o identificador *f* como nome de uma função em sua definição, ela só pode ser chamada através do formato *f(a)*. Isso é uma restrição. Em LPs que tratam funções como valores de primeira classe, a chamada de uma função pode ser feita através de qualquer expressão que retorne uma função. O exemplo seguinte ilustra essa possibilidade em ML:

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

Caso o valor de *difPgVenc* (a diferença entre o dia do pagamento da taxa e o dia de seu vencimento) for um valor positivo, o pagamento é antecipado e a taxa cobrada é calculada pela chamada da função *desconto* aplicada sobre o total de dias antecipados. Caso contrário, o pagamento é feito em atraso e a taxa cobrada é calculada pela chamada da função *multa* aplicada sobre o total de dias atrasados[5.1].

Para se obter uma funcionalidade equivalente em C é necessário utilizar ponteiros para função, o que reduz significativamente a legibilidade e redigibilidade do código, conforme se vê no exemplo 5.8:

```
double (*p)(double);
p = difPgVenc > 0 ? desconto: multa;
```

[5.1] Se o pagamento é feito em dia, a função *multa* retorna o valor da taxa sem multa.

*taxa = (*p) (difPgVenc);*

Exemplo 5. 8 - Chamada Condicional de Função em C

A relação entre chamadas de função e expressões é fortalecida quando se constata que um operador denota uma função. Essa perspectiva fica mais explícita e pode ser melhor observada quando se substitui a notação infixada pela notação prefixada na aplicação de operadores. A tabela 5.1 mostra algumas expressões aritméticas em C (as quais usam a notação infixada) e como seriam suas representações em notação prefixada.

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

Tabela 5. 1 – Operadores em C e Representação Prefixada

Outro modo de observar a relação entre operadores e funções é constatando a correspondência entre os operadores e as assinaturas das funções que denotam. A tabela 5.2 mostra a correspondência entre alguns operadores de JAVA e suas respectivas assinaturas das funções.

Operador	Assinatura da Função
!	[boolean → boolean]
&&	[boolean x boolean → boolean]
*	[char x char → char] [short x short → short] [int x int → int] [long x long → long] [float x float → float] [double x double → double]

Tabela 5. 2 – Operadores em JAVA e suas Assinaturas

Observe na tabela 5.2 que o operador * denota várias funções. Quando isso ocorre, diz-se que o operador é sobrecarregado. Em C, a analogia entre funções e operadores é fraca. Sobrecarga é peculiar a operadores, mas funções não podem ser sobrecarregadas. LPs mais modernas (como C++, ADA, ML e JAVA) reconhecem explicitamente a analogia entre operadores e funções, permitindo que as últimas também sejam sobrecarregadas. C++, ADA e ML vão ainda além que JAVA pois permitem que novas funções sejam associadas aos operadores pelos programadores.

Nessas LPs, <*operando1*> <*operador*> <*operando2*> é exatamente a mesma coisa que <*operador*> *(*<*operando1*>*,* <*operando2*>*)*. Se por um lado, ao evitar separar regras para operadores e funções, essas LPS facilitam o seu aprendizado, por outro lado, a permissão de definição de novos

comportamentos para os operadores tende a tornar a linguagem mais complexa e dificultar seu aprendizado.

5.1.1.9 Expressões com Efeitos Colaterais

O objetivo principal de avaliar uma expressão é retornar um valor. Na maioria das LPs imperativas, contudo, é possível avaliar uma expressão que tenha o efeito colateral de atualizar variáveis. O exemplo seguinte mostra uma expressão (no lado direito da atribuição a *x*) em C que produz como efeito colateral o incremento da variável *c*.

```
x = 3.2 * ++c;
```

C disponibiliza vários operadores de atribuição. Todos eles retornam valores e produzem como efeito colateral a atualização da variável sobre o qual são aplicados.

Normalmente, efeitos colaterais tornam os programas mais difíceis de serem lidos e entendidos. Efeitos colaterais também podem introduzir não determinismo na avaliação de expressões. Por exemplo, na avaliação de uma expressão do tipo <*expressão1*> <*operador*> <*expressão2*>, quando <*expressão1*> afeta <*expressão2*>. Veja o exemplo 5.9 em C:

```
x = 2;
y = 4;
z = (y = 2 * x + 1) + y;
```

Exemplo 5. 9 - Não Determinismo em Expressão com Efeito Colateral em C

No exemplo 5.9, os valores admissíveis para *z* são *9* e *10*. Tudo depende de quando será feita a atribuição a *y* na última linha do exemplo. Se ela for feita antes da avaliação da expressão *y* à direita da soma, o valor de *z* será 10. Caso contrário, o valor de *z* será *9*. Normalmente, a resolução desse tipo de não determinismo não é especificada na LP, deixando para o compilador definir como este tipo de expressão será avaliada. Assim, este tipo de expressão com efeito colateral pode comprometer a portabilidade de programas.

Uma função pode ter em seu corpo uma série de comandos antes de retornar seu valor. É permitido declarar variáveis locais, usar comandos e fazer atribuições. Funções possibilitam a ocorrência de efeitos colaterais através da atualização de variáveis globais ou da passagem de parâmetros por referência. Por exemplo, a função *fgetc(f)* da biblioteca padrão de C produz o efeito colateral de avançar a posição de leitura sobre *f*. Portanto, chamadas de função são expressões que podem produzir efeitos colaterais.

Existem expressões cujo único objetivo é provocar efeitos colaterais. Em C++, por exemplo, o operador *delete* é usado para produzir o efeito colateral de desalocar a memória alocada referenciada por um ponteiro. Este tipo de expressão não retorna qualquer valor (no caso de C++, retorna *void*).

5.1.1.10 Expressões de Referenciamento

Expressões de referenciamento são utilizadas para acessar o conteúdo de variáveis ou constantes[5.2] ou para retornar uma referência a esses objetos (geralmente, o endereço inicial onde esses objetos estão alocados). Normalmente, a expressão utilizada para obter uma referência a uma variável ou constante é também utilizada para acessar o seu conteúdo. Nesses casos, a distinção entre a operação de acesso a conteúdo e a de obtenção de referência é dada pelo local do programa onde a expressão é utilizada, isto é, o seu contexto. A linha de código seguinte ilustra essa característica em C.

```
*q = *q + 3;
```

Observe que a mesma expressão **q* é usada no lado esquerdo e no lado direito da atribuição. A expressão do lado esquerdo retorna uma referência ao objeto apontado por *q*. Já a expressão do lado direito retorna o conteúdo do objeto apontado por *q*.

A forma mais simples de expressão de referenciamento é a de referenciamento direto de variáveis e constantes[5.3]. No exemplo 5.10, o lado esquerdo das atribuições das três linhas do exemplo 5.10 contém expressões de referenciamento que retornam respectivamente o endereço da constante *pi* e os endereços das variáveis *raio* e *perimetro.* Já no lado direito da atribuição da última linha do exemplo 5.10, as expressões de referenciamento *pi* e *raio* retornam o conteúdo desses objetos.

```
const float pi = 3.1416;
int raio = 3;
float perimetro = 2*pi*raio;
```

Exemplo 5. 10 - Referenciamento Direto de Variáveis e Constantes

Outra maneira comum de expressão de referenciamento é usada para acessar estruturas de dados compostas ou anônimas. Os operandos dessas expressões são nomes ou valores que permitem identificar a variável ou constante cuja referência deve ser obtida ou cujo valor deve ser acessado.

[5.2] A operação de acesso a conteúdo é também conhecida como derreferenciamento. Optou-se aqui por utilizar o termo referenciamento para indicar tanto a operação de obtenção de referência quanto a de derreferenciamento.

[5.3] Essas expressões não demandam o uso de um operador explícito.

Por exemplo, para acessar o valor de um elemento de um vetor é necessário ter como operandos o nome do vetor e o índice do elemento. Para realizar esses tipos de operação, LPs normalmente oferecem um conjunto de operadores. A tabela 5.3 mostra os operadores de referenciamento em C, juntamente com o seu significado.

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

Tabela 5. 3 - Operadores de Referenciamento em C

Com exceção do operador &, o qual não pode ser usado para acessar o conteúdo de variáveis (uma vez que não pode ser colocado no lado esquerdo de atribuições), todos os demais operadores da tabela 5.3 são usados para acessar o valor ou retornar uma referência para uma variável ou constante. O exemplo 5.11 ilustra o uso dos operadores da tabela 5.3 para acessar o conteúdo de uma variável e para referenciá-la.

```
p[i] = p[i + 1];
*q = *q + 3;
r.ano = r.ano + 1;
s->dia = s->dia +1;
t = &m;
```

Exemplo 5. 11 - Uso de Operadores de Referenciamento em C

Na primeira linha do exemplo 5.11, *p[i]* retorna o endereço do i-ésimo elemento do vetor *p* e *p[i+1]* retorna o valor do elemento seguinte. Nas três linhas seguintes as expressões no lado esquerdo das atribuições retornam respectivamente o endereço da variável apontada por *q*, o endereço do campo *ano* da estrutura *r* e o endereço do campo *dia* da estrutura apontada por *s.* As mesmas expressões do lado direito das atribuições retornam o conteúdo desses elementos. Por fim, a expressão no lado direito da última linha do exemplo retorna o endereço da variável *m*.

C++ e JAVA possuem ainda uma outra forma de expressão de referenciamento. Essa forma surge com a aplicação do operador *new*. Esse opera-

dor tem por objetivo principal produzir o efeito colateral de alocar memória para um objeto e retorna como resultado uma referência para o objeto alocado. Por retornar uma referência, as expressões que usam *new* são classificadas como de referenciamento.

5.1.1.11 Expressões Categóricas

Expressões categóricas realizam operações sobre tipos de dados. Elas servem para extrair informações a respeito do tipo de uma variável ou constante ou para converter um valor de um tipo para outro. Os operandos de expressões categóricas podem ser um tipo de dados ou um objeto cujo tipo deve ser investigado ou modificado.

C oferece dois operadores para a produção de expressões categóricas: o operador *sizeof* e o operador de conversão de tipo[5.4] *(<tipo>)*. O operador *sizeof* é utilizado para se obter o tamanho em bytes do objeto ou do tipo sobre o qual ele é aplicado. O exemplo 5.12 ilustra o uso desse operador em C.

```
float * p = (float *) malloc (10 * sizeof (float));
int c [] = {1, 2, 3, 4, 5};
for (i = 0; i < sizeof c / sizeof *c; i++) c[i]++;
```

Exemplo 5. 12 –Uso de *sizeof* em Expressões Categóricas

A primeira linha do exemplo 5.12 mostra o operador *sizeof* sendo usado para determinar o tamanho em bytes do tipo *float*. O resultado obtido é usado para alocar um tamanho de memória capaz de armazenar *10* elementos do tipo *float*. Observe com atenção a última linha desse exemplo. Nessa linha o operador *sizeof* é usado tanto para obter o tamanho do vetor *c* quanto para obter o tamanho do elemento de *c*. Isso permite que essa linha não seja alterada quando for necessário alterar o tamanho do vetor *c*.

O operador de conversão de tipo *(<tipo>)* é utilizado para converter um valor de um tipo para outro. Seus operandos são o valor (ou variável) que se deseja converter e o tipo para o qual o valor deve ser convertido. O exemplo 5.13 ilustra o uso desse operador em C.

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

Exemplo 5. 13- Operador de Conversão de Tipo

Na última linha do exemplo 5.13, o valor do tipo inteiro *int*, armazenado em *num,* é convertido para o valor correspondente no tipo *float*. Desse modo, o valor atribuído a *f* será o resultado da operação de divisão real

[5.4] O operador de conversão de tipo é também conhecido como operador de cast.

pois um dos seus operandos se tornou um valor *float*. C++ oferece vários outros operadores de conversão de tipos. Esses operadores serão discutidos no capítulo 7.

C++ e JAVA oferecem outra forma de expressão categórica, a qual possibilita verificar dinamicamente se uma determinada variável ou constante pertence a um certo tipo. C++ utiliza o operador *typeid* e JAVA utiliza o operador *instanceof* para realizar esse tipo de operação. O exemplo 5.14 mostra o uso do operador *instanceof* em JAVA.

```
Profissao p = new Engenheiro ( );
if (p instanceof Medico) then
        System.out.println ("Registre-se no CRM");
if (p instanceof Engenheiro) then
        System.out.println ("Registre-se no CREA");
```

Exemplo 5. 14 - Uso de instanceof em JAVA

No exemplo 5.14, caso *p* fosse uma referência para um objeto do tipo *Medico*, a expressão categórica na segunda linha do exemplo seria satisfeita e uma mensagem seria apresentada indicando que o objeto referenciado por *p* deveria se registrar no CRM (Conselho Regional de Medicina). Como não é esse o caso, essa mensagem não será apresentada. Já a expressão categórica da penúltima linha do exemplo será satisfeita e o programa apresentará uma mensagem indicando que o objeto deve se registrar no CREA (Conselho Regional de Engenharia e Arquitetura).

5.1.2 Avaliação de Expressões Compostas

Expressões compostas sempre envolvem duas ou mais operações. A ordem de avaliação das operações de uma expressão composta pode influenciar completamente o resultado obtido. Por exemplo, a expressão *3+2*5* pode resultar no valor *25* (caso a operação de soma seja efetuada primeiro) ou no valor *13* (caso a primeira operação seja a multiplicação).

Os conceitos importantes para a determinação da ordem de avaliação de expressões compostas são as regras de precedência, associatividade e curto circuito dos operadores e as regras de precedência entre operandos.

5.1.2.1 Precedência de Operadores

Normalmente, os operadores de uma LP são ordenados por grau de precedência. Operações cujos operadores possuem maior precedência são realizadas antes das operações relacionadas aos operadores de menor precedência.

Caso não exista precedência entre operadores na LP (APL e SMALLTALK não possuem), certas expressões compostas são avaliadas em uma ordem pouco natural. Por exemplo, expressões compostas aritméticas não são avaliadas na mesma sequência usada pela matemática básica. Assim, uma operação de soma pode preceder uma operação de multiplicação em uma expressão composta como *3+2*5*. Isso certamente dificulta o entendimento do programa. Por outro lado, a existência de graus de precedência entre operadores exige que o programador se lembre das suas precedências na hora de construir uma expressão composta, o que pode provocar enganos.

Parênteses podem ser usados para garantir que a avaliação ocorra na ordem desejada em situações nas quais o programador não se lembra da precedência dos operadores ou precisa garantir uma ordem de avaliação distinta daquela determinada pelos graus de precedência. De fato, o uso de parênteses pode inclusive dispensar a LP de definir o conceito de precedência entre operadores. Quando se deseja escrever uma expressão composta avaliada segundo uma ordem natural, basta usar os parênteses.

Embora o uso de parênteses possa até facilitar o entendimento de certas expressões, o seu uso excessivo pode baixar a redigibilidade (tem de se fechar muitos parênteses e escrever mais) e legibilidade (a expressão se torna muito longa e dificil de ser lida) da expressão composta. Outro problema com parênteses é impedir que o compilador realize certos tipos de otimização de código que demandam a alteração na ordem de avaliação das expressões compostas.

A escolha inadequada das precedências entre operadores numa LP pode afetar a redigibilidade de programas e provocar erros de programação. Por exemplo, em PASCAL, os operadores relacionais têm menor precedência que os lógicos. Como resultado, expressões bem formadas que combinam estes operadores requerem o uso de parênteses para alterar a ordem de precedência. O exemplo 5.15 ilustra esse problema em PASCAL.

```
/* if a > 5 and b < 10 then */
if (a > 5) and (b < 10) then a := a + 1;
```

Exemplo 5. 15 – Escolha Inapropriada de Precedência de Operadores em PASCAL

No exemplo 5.15, a linha comentada geraria um erro de compilação, uma vez que a ordem de precedência impõe ao compilador a geração de código para avaliar primeiro a expressão *5 and b*. Como nenhum dos dois operandos é do tipo booleano, essa expressão não é válida. Nesse caso, a solução é incluir parênteses, tal como na linha final do exemplo. Essa exigência, contudo, afeta a redigibilidade do programa torna tedioso o trabalho do programador.

A tabela 5.4 mostra a ordem de precedência dos operadores de C [KERNIGHAN & RITCHIE, 1989]. Os parênteses são usados para designar o operador de chamada de função *( )* e o operador de conversão de tipos *(<tipo>)*. Observe que a maior precedência é a de grau 1. Note ainda que os operadores –, * e & com grau 2 de precedência são os unários. Cabe destacar também que os operadores relacionais tem maior precedência que os operadores booleanos binários.

Precedência	Associatividade	Operadores
1	esquerda para a direita	*( ) [ ] –>*
2	direita para a esquerda	*! ~ ++ — + – (<tipo>) * & sizeof*
3	esquerda para a direita	** / %*
4	esquerda para a direita	*+ –*
5	esquerda para a direita	*<< >>*
6	esquerda para a direita	*< <= > >=*
7	esquerda para a direita	*== !=*
8	esquerda para a direita	*&*
9	esquerda para a direita	*^*
10	esquerda para a direita	*|*
11	esquerda para a direita	*&&*
12	esquerda para a direita	*||*
13	direita para a esquerda	*?:*
14	direita para a esquerda	*= += –= *= /= %= &= |= ^= <<= >>=*
15	esquerda para a direita	*,*

Tabela 5. 4 - Tabela de Precedência e Associatividade de Operadores em C

5.1.2.2 Associatividade de Operadores

Regras de associatividade de operadores são usadas quando a LP não define regras de precedência entre operadores ou quando operadores adjacentes da expressão composta têm a mesma precedência. Nesses casos, a regra de associatividade é quem determina qual operador será avaliado primeiro.

Na maioria das LPs a regra de associatividade de operadores usada é a avaliação da esquerda para direita. O exemplo 5.16 mostra a aplicação dessa regra em duas expressões compostas em C.

```
x = a + b – c;
y = a < b < c;
```

Exemplo 5. 16 - Associatividade de Operadores

Na primeira linha do exemplo 5.16, a soma dos valores de *a* e *b* é feita primeiro e do seu resultado é subtraído o valor de *c*. Na segunda linha o valor de *a* é comparado com o valor de *b* e o resultado é comparado com o valor de *c.* Note que essa expressão composta não avalia se os valores de *a*, *b* e *c* são crescentes. De fato, essa expressão só é válida porque as expressões relacionais de C retornam um valor inteiro. Assim, o valor retornado pela primeira comparação (*0* ou *1*) é comparado com o valor de *c*.

Embora a maior parte dos operadores de uma LP siga a regra geral de associatividade, frequentemente existem operadores que não a obedecem. Isso ocorre em C, como pode ser observado na tabela 5.4. No entanto, boa parte dos operadores de C que adotam regra de associatividade invertida (isto é, da direita para a esquerda) não teriam sentido se definidos de outra forma. O exemplo 5.17 ilustra o uso de alguns desses operadores.

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

Exemplo 5. 17 - Associatividade da Direita para a Esquerda em C

Perceba que, para avaliar ***p*, *!!x* e *a = b = c*, é absolutamente necessário associar os operadores da direita para a esquerda. Nem sempre isso ocorre dessa maneira. Em C, a expressão *!x++* pode gerar dúvidas no programador quanto a ordem de avaliação. Em FORTRAN, o operador de exponenciação é associativo da direita para a esquerda, embora os demais operadores sejam da esquerda para a direita. Observe que nesse caso esse operador poderia ser associativo da esquerda para a direita. Embora essa opção seja decorrente da forma como essa operação é realizada na matemática, programadores inadvertidos podem cometer erros por conta dessa opção.

Algumas LPs podem fazer opções controvertidas com relação a associatividade dos operadores. APL, por exemplo, além de não estabelecer qualquer precedência entre os operadores, ainda opta por avaliar sempre os operadores da direita para a esquerda. Na linha de código APL apresentada a seguir a subtração será realizada antes da divisão.

$$X = Y \div W - Z$$

Alguns compiladores podem usar situações onde não existe precedência entre os operadores da expressão para fazer otimizações de código. Por exemplo, na seguinte linha de código C, as funções *f*, *g* e *h* retornam inteiros e *x* é uma variável inteira:

```
x = f() + g() + h();
```

Nesse caso a ordem de avaliação da esquerda para a direita das funções poderia ser alterada para somar *f* com *h* e depois somar o resultado com *g*. Contudo, isso pode gerar problemas sérios. Por exemplo, se *f* e *h* retornam inteiros positivos muito grandes e *g* retorna um inteiro negativo muito grande, a soma de *f* com *h* pode provocar overflow, enquanto que a soma de *f* com *g* não provoca. Situações como essa devem ser evitadas através do uso de parênteses para garantir a ordem de avaliação de operadores.

5.1.2.3 Precedência de Operandos

O exemplo 5.9 mostra uma situação em C na qual ocorre não determinismo por causa do uso de expressões com efeito colateral. Esse não determinismo ocorre porque C não especifica a ordem na qual os operandos de um operador devem ser avaliados (as únicas exceções a essa regra são os operadores *&& || ?: ,*). Assim, o compilador é quem decide qual operando será avaliado primeiro. Como a avaliação de um operando pode afetar ou não a avaliação do outro, surge o não determinismo. É importante destacar que esse não determinismo pode ocorrer em situações comuns. O exemplo seguinte mostra uma dessas situações de não determinismo. Nesse exemplo, não se sabe se o índice do vetor será o valor de *i* antigo ou o novo.

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

Exemplo 5. 18 - Não Determinismo em C (extraído de [KERNIGHAN & RITCHIE, 1989])

C não fixa intencionalmente essa ordem para que o compilador possa explorar adequadamente a arquitetura específica do computador e poder gerar código mais eficiente.

Uma forma de evitar o não determinismo na avaliação dessas expressões é estabelecer regras de precedência para a avaliação dos operandos. Por exemplo, JAVA estabelece que operandos são sempre avaliados da esquerda para a direita. Assim, implementações em JAVA dos exemplos 5.9 e 5.18 são determinísticas. No caso do exemplo 5.9, o valor atribuído a *z* é *10* e, no exemplo 5.18, o valor de *i* é o antigo. Contudo, a adoção de precedência entre operandos inibe certos tipos de otimização de código, impedindo ao código executável se tornar mais eficiente.

5.1.2.4 Avaliação de Expressões Compostas com Curto Circuito

A avaliação de expressões em curto circuito ocorre quando o resultado é determinado antes que todos os operadores e operandos tenham sido avaliados.

A linha de código seguinte mostra uma situação potencial em C na qual poderia ser empregada a avaliação com curto circuito.

```
z = (x - y) * (a + b) * (c – d);
```

Quando o valor de um dos operandos de uma multiplicação resulta em zero (por exemplo, quando o valor de *x* é igual ao valor de *y*), não seria necessário avaliar o outro operando para saber o resultado da expressão composta. Como este tipo de curto circuito não ocorre frequentemente, não vale a pena incluí-lo em LPs visto que ele tornaria o código mais eficiente somente quando um dos operandos da multiplicação resultasse no valor zero, mas em todos os outros casos ele embutiria testes desnecessários, reduzindo assim a eficiência do código gerado.

Avaliações em curto circuito são muito usadas para avaliar as expressões booleanas binárias de conjunção e disjunção. Quando o primeiro operando avaliado na conjunção não é satisfeito, não existe necessidade de avaliar o outro operando. O mesmo ocorre na disjunção, desde que o operando avaliado primeiro seja satisfeito. Observe que nesses casos também é necessário incluir um teste adicional ao código para realizar curto circuito. Contudo, como essas condições ocorrem com bastante frequência, vale a pena utilizar curto circuito.

Programas em JAVA usam curto circuito em situações tais como a mostrada no exemplo 5.19.

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

Exemplo 5. 19 - Uso de Curto Circuito em JAVA

Esse trecho de programa é usado para procurar a posição do valor *v* no vetor *a*. Observe o uso do curto circuito quando o valor *v* não está presente no vetor *a*. Nesse caso, o valor de *i* se igualará a *n* e a primeira condição da expressão booleana não será satisfeita, não havendo necessidade de avaliar a segunda condição. Caso não se utilizasse curto circuito, haveria um erro de execução na avaliação da segunda condição pois o programa tentaria acessar uma posição inexistente no vetor *a* (a posição de índice *n*).

Em PASCAL, os operadores binários booleanos não empregam curto circuito. Isso impede que os programadores criem código como o do exemplo 5.19. Cabe ressalvar que alguns compiladores PASCAL permitem ao programador optar por gerar código com ou sem avaliação em curto circuito. Esta postura é mais flexível, embora não permita o uso dos dois tipos de avaliação simultaneamente e possa comprometer a portabilidade do código.

Algumas LPs adotam operadores booleanos específicos para a avaliação com e sem uso de curto circuito. Esse é o caso de ADA e JAVA. Os operadores booleanos *and* e *or* de ADA e *&* e | de JAVA não usam curto circuito. Já os operadores booleanos *and then* e *or else* de ADA e *&&* e || de JAVA usam. Essa é uma solução flexível e geral, embora implique na existência de um maior número de operadores, o que torna a LP mais complexa.

Os únicos operadores booleanos de C e C++ são *&&* e ||, os quais usam curto circuito. Contudo, os operadores binários *&* e | não usam e podem ser usados para avaliar expressões booleanas sem o uso de curto circuito. Para tanto, é preciso levar em conta que uma expressão relacional retorna o valor zero, quando não é satisfeita, e o valor um, em caso contrário. Assim, a aplicação dos operadores binários | e *&* a operandos que são expressões relacionais geram os mesmos comportamentos dos respectivos operadores booleanos. Observe, no entanto, que o operador binário de conjunção pode não funcionar a contento caso um dos operandos da expressão binária não seja uma expressão relacional.

A avaliação com curto circuito pode gerar programas difíceis de serem entendidos quando em associação com efeitos colaterais. O trecho de código em C apresentado a seguir não incrementa o valor de *i* quando $b < 2*c$. Nesse caso, o elemento de *a* incrementado é o correspondente ao valor de *i* sem incremento. Caso contrário, se a segunda condição da disjunção for satisfeita, o elemento de *a* incrementado correponde ao do valor de *i* com incremento. Por outro lado, se a segunda condição também não for satisfeita, nenhum elemento de *a* será incrementado.

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

5.2 Comandos

Comandos são instruções do programa que tem o objetivo de atualizar as variáveis ou controlar o fluxo de controle do programa. Comandos são característicos de linguagens imperativas. De fato, o nome imperativo advém da visão que programas nessas linguagens usam comandos para determinar as computações a serem realizadas pelo computador. Em outras palavras, a linguagem é imperativa porque os programas comandam o computador.

Os comandos podem ser primitivos ou compostos. Enquanto comandos primitivos (por exemplo, um comando de atribuição) em geral não podem ser subdivididos em outros comandos, comandos compostos normalmente possuem um ou mais subcomandos em seu escopo de ação (por exemplo, um comando de repetição possui pelo menos um subcomando que será repetido para alterar o estado da condição de parada do comando).

Para uma LP imperativa ser suficientemente expressiva ela necessita de pelos menos três tipos de comandos: um comando de atribuição para permitir a atualização de variáveis; um comando de seleção para permitir a existência de caminhos alternativos no fluxo de controle do programa e um comando de desvio do fluxo de controle para permitir a realização de repetições de comandos. É claro que uma LP com apenas esses tipos de comandos seria extremamente limitada. De fato, até mesmo linguagens consideradas de baixo nível possuem uma maior variedade de comandos do que a listada nesse parágrafo. Assim, normalmente, as LPs oferecem várias alternativas para esses tipos de comandos, bem como alguns outros comandos complementares.

5.2.1 Tipos de Comandos

Comandos de LPs imperativas podem ser classificados de acordo com a natureza da operação que realizam. Eles podem ser divididos nas seguintes categorias fundamentais [WATT, 1990]: atribuições, comandos seqüenciais, comandos colaterais, comandos condicionais, comandos iterativos, chamadas de procedimento e comandos de desvio incondicional.

5.2.1.1 Atribuições

Em sua forma geral, o comando de atribuição envolve o uso de um símbolo que designa o comando, uma expressão que produz uma referência à variável cujo valor será atualizado e uma expressão que produz como resultado um valor a ser armazenado nessa variável.

Cada LP usa um símbolo próprio para designar a operação de atribuição. C, C++ e JAVA usam o símbolo =. ADA, PASCAL e MODULA-2 usam *:=* porque o símbolo = costuma causar problemas no uso e aprendizado da linguagem.

Uma das dificuldades enfrentadas pelo aprendiz é a tendência em confundir o comando de atribuição com a operação de igualdade da matemática. Assim, um comando do tipo $i = i + 1$ tende a ser entendido como uma equação sem solução ao invés de um comando de atribuição.

Essa dificuldade de entendimento é agravada pelo fato da variável *i* designar dois conceitos distintos nesse comando (respectivamente, uma referência à variável e o conteúdo dessa variável). Cabe ressalvar, contudo, que esse problema não é relacionado com o símbolo usado e sim com o significado de uma atribuição. ML reconhece esse problema e obriga a utilização de um operador de derreferenciamento (*!*) para acessar o conteúdo da variável. Assim, um comando de atribuição equivalente em ML tem o seguinte formato: *i := !i + 1*.

Outra dificuldade relacionada com uso do símbolo = para designar uma atribuição é a confusão da operação de atribuição com a operação de comparação de igualdade. Esse problema se agrava quando o símbolo = também é usado para designar a operação de igualdade, tal como em PL-1, ou quando se pode usar uma atribuição no local onde se espera uma comparação, como é o caso de C e C++. A linha seguinte de código C válido mostra o uso de uma atribuição onde deveria haver uma comparação. Essa linha certamente é um erro de programação ou de digitação pois, dessa maneira, a condição é sempre satisfeita e o valor de *a* é sempre incrementado de *3*. Isso torna inócua a existência desse comando *if*.

if (a = 10) a += 3;

Existem vários tipos de comandos de atribuição [SEBESTA, 1999]. Cada um desses tipos é apresentado a seguir:

a. Atribuição simples: é o tipo mais comum de atribuição. Nesse tipo o resultado de uma expressão é atribuído a uma variável, tal como no próximo exemplo em C.

   *a = b + 3 * c;*

b. Atribuição múltipla: ocorre quando se atribui o mesmo valor a diversas variáveis. Por exemplo, em C, o seguinte comando atribui o valor zero para as variáveis *a* e *b*.

   a = b = 0;

c. Atribuição condicional: atribui o valor de uma expressão ao resultado de uma expressão condicional. Por exemplo, na linha seguinte de código ML, o valor *2* será atribuído a variável *a* ou *b*, dependendo da avaliação da expressão *a<b*.

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

d. Atribuição composta: permite a combinação de operadores binários com o comando de atribuição. Nos exemplos em C, mostrados a seguir, a variável *a* tem seu valor incrementado de *3*, multiplicado por *3* e modificado para o resultado da sua conjunção com *3*, respectivamente.

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

e. Atribuição unária: permite a atribuição ser realizada com um único operando (o outro é implícito). Em todos os exemplos seguintes em C a variável *a* tem seu valor incrementado ou decrementado de *1*.

   ++a; *a++;* *--a* *a--;*

   A distinção entre a posição prefixada e posfixada dos operadores ++ e -- se refere ao valor retornado pela atribuição. Na primeira linha do

código C seguinte, o valor corrente de *a* é atribuído a *b* e, depois, incrementado. Na segunda linha ocorre o inverso, isto é, o valor de *a* é incrementado e, somente depois disso, atribuído a *b*.

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

f. Atribuição expressão: o comando de atribuição retorna o valor atribuído. Todos os tipos de atribuição em C também são exemplos de atribuição expressão. Esse tipo de atribuição possibilita ao programador criar atalhos interessantes em certas situações, tal como ilustrado no seguinte código C.

```
while (( ch = getchar ( ) ) != EOF ) { printf("%c", ch); }
```

Por outro lado, a atribuição expressão em C pode incentivar o programador a criar código pouco legível e, quando associada com a ausência de tipo booleano, tende a provocar erros que não podem ser detectados pelo compilador (tal como aquele em que se coloca uma atribuição onde se espera uma comparação). JAVA elimina esse problema só permitindo o uso de expressões booleanas nessas situações.

5.2.1.2 Seqüenciais

O modo mais comum de fluxo de controle existente em toda LP imperativa é a composição seqüencial de comandos. O comando antecedente na seqüencia é executado antes do subsequente. Normalmente, o conceito de bloco é usado na implementação dos comandos seqüenciais. Nesses casos, um bloco permite que uma série de comandos seqüenciais sejam abstraídos em um único comando. O exemplo 5.20 mostra como blocos são usados em C para compor comandos seqüenciais:

```
{
  n = 1;
  n += 3;
  if (n < 5){
        n = 10;
       m = n * 2;
  }
}
```

Exemplo 5. 20- Comandos Seqüenciais em C

Algumas LPs não usam diretamente o conceito de blocos para agrupar comandos seqüenciais. Nessas LPs, o agrupamento de comandos seqüenciais é embutido dentro dos outros comandos da LP. Exemplos de LPs que adotam essa postura são ADA e MODULA-2.

5.2.1.3 Colaterais

Esse tipo de comando é pouco comum em LPs imperativas. Nesses comandos não existe uma ordem prévia para a execução dos comandos. No exemplo seguinte as variáveis *a* e *b* são atualizadas independentemente e a ordem de execução é irrelevante, isto é, o(s) processador(es) pode(m) executar os dois comandos na ordem que for mais conveniente.

```
a = f(x), b = g(y);
```

Ao se programar numa LP com comandos colaterais é necessário ter cuidado para evitar o uso inadequado desses comandos. No exemplo seguinte o valor de *a* depende da ordem de execução do comando colateral.

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

Os valores possíveis de *a* são *4*, se os comandos colaterais forem executados na ordem da esquerda para direita, ou *3,* se na ordem inversa, ou mesmo *1*. Esse último valor é obtido através da sequência: o valor *0* é atribuído a variável *a*, esse valor é utilizado para avaliar a expressão *a + 1*, o valor *3* é atribuído a variável *a*, o resultado da expressão *a + 1* (no caso, o valor *1)* é atribuído a variável *a*.

Uma computação é determinística quando se pode prever a ordem de execução dos comandos. De outra maneira, a computação é não determinística. Comandos colaterais são não determinísticos. Uma computação não determinística pode ter um efeito previsível, embora a ordem de execução dos comandos não seja conhecida (nesse caso, ela é considerada efetivamente determinística). Um comando colateral é efetivamente determinístico se nenhum subcomando pode acessar uma variável atualizada por outro subcomando. O exemplo 5.21 ilustra o uso de comandos colaterais em ML. Não se pode saber previamente a ordem em que as definições de *altura*, *largura* e *comprimento* serão realizadas.

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

Exemplo 5. 21 - Comando Colateral em ML

Cabe destacar que ML impede o uso de um identificador definido em um dos subcomandos em qualquer dos outros subcomandos. Assim, seria ilegal adicionar ao exemplo 5.21 a seguinte linha de código:

```
and volume = altura * largura * comprimento
```

Comandos colaterais são fundamentais em LPs destinadas ao processamento paralelo de programas. Nessas LPs, os comandos colaterais indi-

cam quais comandos podem ser executados paralelamente em diferentes processadores.

5.2.1.4 Condicionais

Um comando condicional (também conhecido como comando de seleção) permite a especificação de caminhos alternativos para o fluxo de controle do programa. Ele possui um número de subcomandos dos quais exatamente um é escolhido para ser executado.

A maioria das LPs fornece três tipos de comandos de seleção. Cada um desses tipos é apresentado a seguir.

5.2.1.4.1 Seleção de Caminho Condicionado

Esse comando permite que um trecho de programa seja executado se determinada condição é satisfeita. O comando *if* de C pode atuar dessa maneira.

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

O trecho entre chaves só é executado quando a condição é verdadeira.

5.2.1.4.2 Seleção de Caminho Duplo

Esse comando permite que exista uma condição para a escolha entre dois trechos alternativos de programa. O comando *if* de C também pode atuar dessa maneira.

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

O primeiro trecho entre chaves somente é executado quando a condição é verdadeira, e o segundo, somente quando ela é falsa.

Em muitas LPs, tais como C e PASCAL, ocorre um problema de ambigüidade sintática na escrita de comandos condicionais duplos aninhados. O exemplo 5.22 ilustra esse problema em C.

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

Exemplo 5. 22 - Problema com Aninhamento em Comando Duplo de Seleção

Não existe no comando acima um distinção sintática indicando a qual *if* o *else* pertence. De fato, existe ambigüidade apesar da semântica sugerida pela disposição do texto do código. C adota regra que o *else* se refere ao *if*

mais interno. Por ser uma decisão arbitrária, ela é difícil de ser memorizada e fácil de provocar erro. Além disso, para se conseguir a semântica indicada pela disposição do texto, é estritamente necessário colocar marcadores de bloco no comando *if* mais externo (tal como ilustrado no exemplo 5.23.

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

Exemplo 5. 23- Uso de Marcadores de Bloco em Comando Duplo de Seleção em C

O problema de C e PASCAL acontece pela falta de um símbolo específico indicador de fechamento de comando condicional. MODULA-2 e ADA possuem palavras especiais de fechamento de comandos condicionais. O exemplo 5.24 ilustra essa solução em ADA.

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

Exemplo 5. 24 - Terminação de Comandos de Seleção em ADA

5.2.1.4.3 Seleção de Caminhos Múltiplos

Esse comando permite que exista uma escolha entre várias alternativas de execução do programa conforme o resultado de uma expressão. O exemplo 5.25 ilustra como o comando *switch* de C pode atuar dessa maneira.

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

```
}
```

Exemplo 5. 25 - Comando de Caminhos Múltiplos em C

Note no exemplo 5.25 que tanto a nota *9* quanto a nota *10* implicam na impressão da mensagem "*Muito Bom!!!*". Observe também a necessidade do uso do comando *break*. Sem ele, todas as quatro mensagens seriam impressas no caso da nota ser *9* ou *10*.

Existem variações significativas nos comandos condicionais de caminhos múltiplos das diversas LPs. Estas variações vão desde o nome do comando (*GOTO* em FORTRAN, *case* em ADA e PASCAL e *switch* em C) até o funcionamento do comando (enquanto no *case* de ADA e PASCAL a execução vai automaticamente para o fim do comando condicional após a execução de um caminho, no *switch* de C é necessário inserir o comando *break* ao final do caminho para se ter o mesmo comportamento).

LPs como C e PASCAL restringem as expressões usadas como condição e seus possíveis resultados a constantes de tipos ordinais. Quando a seleção deve ser feita com base em expressões lógicas devem ser usados comandos *if* aninhados. O exemplo 5.26 mostra esse tipo de abordagem em C.

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

Exemplo 5. 26 - Caminhos Múltiplos com *if* Aninhados

Algumas LPs, como ADA, MODULA-2 e FORTRAN-90 reconhecem a importância desse tipo de construção e usam um único comando *if* com um mecanismo específico (*elsif*) para tratar esses casos.

5.2.1.5 Iterativos

Um comando iterativo (também conhecido como comando de repetição[5.5]) permite a especificação de ciclos no fluxo de controle do programa. Iterações são típicas de LPs imperativas. Um comando iterativo possui um subcomando (o seu corpo) o qual é executado repetidamente até que a satisfação de algum tipo de condição determine o seu fim.

Comandos iterativos podem ter o número de repetições definido previamente (antes da primeira execução do corpo do comando) ou não.

[5.5] Comandos de repetição são frequentemente designados pelo termo loop.

5.2.1.5.1 Número Indefinido de Repetições

Esse comando é usado quando o número de iterações não é determinado previamente. Em geral, existem 2 tipos desses comandos em LPs: um com pré-teste e outro com pós-teste. O exemplo 5.27 ilustra esses dois comandos em C.

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

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

Exemplo 5. 27 - Comando Iterativo com Número de Repetições Indefinido

No exemplo 5.27 os dois trechos com comando de repetição calculam o fatorial *f* de um número natural *x*. No comando com pré-teste (o comando *while*), o corpo da repetição só é executado se a condição de parada é verdadeira pelo menos uma vez. No comando com pós-teste (o comando *do-while*), o corpo da repetição é sempre executado pelo menos uma vez.

Embora atendam a maior parte dos casos nos quais comandos com número indefinido de repetições são necessários, em algumas situações pode ser conveniente interromper a repetição após algum comando interno do corpo da repetição. O exemplo 5.28 mostra o uso dos comandos iterativos de C em uma dessas situações.

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

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

Exemplo 5. 28 - Situação na qual Comandos com Pré ou Pós-Teste não são os mais Indicados

No exemplo 5.28 os dois trechos com comando de repetição calculam a soma *s* de vários números inteiros positivos *n* até que seja lido um número inteiro não positivo. Enquanto no comando *while* é necessário repetir o uso dos comandos *printf* e *scanf* para a leitura de *n* (impactando a redigibilidade do código), no comando *do-while* é necessário repetir a verificação se o número *n* é inteiro positivo (impactando a redigibilidade e, principalmente, a eficiência do código).

Além do tipo de situação ilustrado no exemplo 5.28, muitas vezes o programador necessita incluir várias condições de saída da iteração em pon-

tos distintos do corpo da repetição. As LPs oferecem comandos de escape (serão vistos na seção 5.2.1.7), os quais podem ser combinados com comandos de repetição e seleção, para atender esses tipos de demandas.

5.2.1.5.2 Número Definido de Repetições

Esse comando é usado quando o número de iterações é determinado previamente. Esse tipo de iteração é caracterizado pelo uso de uma variável de controle. O corpo da repetição é executado com a variável de controle assumindo cada valor de uma seqüência pré-determinada de valores.

Em geral, além da variável de controle, esses comandos possuem como elementos as expressões que determinam os valores inicial e final da variável de controle e, opcionalmente, a expressão que determina o valor da variação da variável de controle entre um ciclo e outro da repetição. Todas essas expressões devem ser avaliadas antes da execução do primeiro ciclo da repetição. O exemplo 5.29, em MODULA-2, utiliza todos esses elementos para somar os números inteiros múltiplos de *j* no intervalo compreendido entre a *j*-ésima dezena e a dezena seguinte.

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

Exemplo 5. 29 - Comando Iterativo com Número de Repetições Definido em MODULA-2

Observe no exemplo 5.29 que *i* é a variável de controle, *10*j* é a expressão usada para determinar o valor inicial de *i*, *10*(j+1)* é a expressão usada para determinar o valor final de *i*, e *j* é a expressão que determina o valor da variação da variável de controle. Note também que os valores a serem assumidos por *i* em cada repetição podem ser conhecidos antes da realização do primeiro ciclo. Por exemplo, se o valor de *j* é *4*, os valores assumidos por *i* serão *40*, *44* e *48*.

Existem algumas posições consensuais a respeito do comando iterativo com número de repetições definida. A realização do teste de parada deve ser feita antes da execução do corpo da repetição para cada valor da variável de controle. O número de repetições a ser realizada deve ser avaliado antes do começo da repetição e permanecer fixo. Deve-se ainda impedir múltiplas entradas no corpo da repetição (por exemplo, através do uso de um comando de desvio do fluxo do controle).

Contudo, também existem grandes variações nas características desse tipo de comando entre as diversas LPs. Variações ocorrem nos tipos possíveis da variável de controle e no seu escopo de visibilidade, na possibilidade de alteração da variável de controle e dos outros elementos do comando

no corpo da repetição e na possibilidade de existência da expressão definidora da variação do valor da variável de controle.

Em ADA, a variável de controle somente pode ser de um tipo primitivo discreto (inteiro ou intervalo de inteiros ou enumeração) e seu escopo é restrito ao corpo da repetição. Isso significa que não é possível referenciar a variável de controle fora do comando de repetição. ADA não permite que a variável de controle tenha seu valor alterado por um comando do corpo da repetição, mas as variáveis usadas para especificar o intervalo da repetição podem ser alteradas, uma vez que as expressões só são avaliadas uma única vez no início da execução da repetição. Caso o intervalo de variação determinado para a variável de controle seja nulo, o corpo da repetição não é executado. A variável de controle de ADA deve assumir todos os valores do intervalo de variação especificado, uma vez que não há possibilidade de se especificar a variação de valor entre um ciclo e outro. No entanto, ADA possibilita que a ordem de atribuição dos valores seja normal ou reversa.

Já FORTRAN e PASCAL consideram a variável de controle como uma variável ordinária cujo valor é atribuído sucessivamente após a execucão de cada corpo da repetição. Essa postura não esclarece as questões sobre o valor da variável de controle após o encerramento do comando de repetição e se ela pode ser alterada dentro dele. Isso é deixado para o implementador da LP, o que pode comprometer a portabilidade dos programas.

MODULA-2 determina que a variável de controle deve ser de um tipo primitivo discreto e a expressão que determina o tamanho do passo tem que ser inteira. A variável de controle não pode ser um parâmetro passado por referência, um campo de agregado ou uma variável importada. Quando termina a repetição o valor da variável de controle é indefinido. O corpo da repetição não pode alterar a variável de controle nem as variáveis utilizadas para definir as expressões que determinam o valor inicial e final da iteração.

C possui o comando *for*, que embora aparente ser um comando iterativo de repetição definida, também pode ser usado como comando de repetição indefinida. Esse comando de C estabelece quatro regiões nas quais podem ser incluídos subcomandos. O trecho de inicialização, onde são inicializadas uma ou mais variáveis de controle, é executado apenas uma vez no início da execução da repetição. O trecho de teste é executado antes de cada iteração. O trecho de avanço é executado depois de cada iteração. Por fim, o corpo da repetição contém os comandos que são executados repetidamente. O exemplo 5.30 ilustra esse comando em C calculando a diferença *dif* entre a soma de todos os números divisíveis por *2* e a soma de todos os números divisíveis por *3* do vetor *a*:

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

Exemplo 5. 30 - Comando *for* de C

É importante destacar as peculiaridades do comando *for* de C. Cada região do comando pode ser composta por uma seqüência de expressões (o valor retornado pela região é o valor da última expressão). Todas as quatro regiões do *for* são opcionais. Não existe uma variável de controle explícita. É possível misturar condições lógicas e de contagem no teste de parada e todas as variáveis envolvidas podem ser alteradas dentro do corpo da repetição. O exemplo 5.31 ilustra várias dessas possibilidades.

```
for (i = 10 * j, s = 0; i <= 10 * (j + 1); s += i++);
for (i = 0, s = 0; i <= n && s < 101 && a[i] > 0 ; ) s += a[i++];
for (;;);
```

Exemplo 5. 31 - Possibilidades Oferecidas por Comando *for* de C

A primeira linha do exemplo 5.31 cumpre a mesma funcionalidade realizada pelo exemplo 5.29 em MODULA-2. Observe que as variáveis *i* e *s* são inicializadas e incrementadas nas regiões de inicialização e avanço do próprio comando *for*. Observe ainda que esse comando não possui corpo.

A segunda linha do exemplo 5.31 é usada para somar os elementos do vetor *a* até que se chegue ao final desse vetor ou a soma seja superior a *100* ou seja encontrado um número não positivo no vetor. Note que a condição de parada do *for* é uma expressão booleana que combina várias expressões relacionais, a região de avanço não possui qualquer comando e que as variáveis *i* e *s* têm seu valor alterado no corpo do comando.

A terceira linha do exemplo 5.31 contém um comando *for* no qual todas as regiões não possuem comandos. Embora esse comando seja sintaticamente válido em C, ele não é correto do ponto de vista semântico, uma vez que o programa entraria em uma repetição infinita.

Cabe ressalvar ainda que comandos tais como os das duas primeiras linhas do exemplo 5.31 não devem ser utilizados de modo geral pois, embora ofereçam uma boa redigibilidade, eles comprometem muito a legibilidade dos programas.

O comando *for* de C++ se difere do de C apenas por poder incluir a definição das variáveis de controle dentro do próprio comando. Contudo, a variável definida é visível a partir do comando *for* até o final do bloco onde o *for* é definido. Já em JAVA, o escopo de visibilidade da variável é limitado pelo próprio comando *for*.

Para finalizar a discussão sobre o comando iterativo com número de repetições definida, cabe ressaltar a não existência de uma razão fundamental para a seqüência de controle ser restrita a uma progressão (ou regressão) sobre tipos primitivos discretos. Um tipo mais geral desse tipo de comando deve permitir a iteração sobre qualquer coleção de elementos. Os comandos de repetição existentes nas LPs abordadas anteriormente seriam um caso particular no qual a coleção é formada pelos elementos especificados no intervalo entre o valor da expressão inicial e final do comando. Tal postura facilitaria muito a programação pois permitiria percorrer listas de valores, arquivos, conjuntos e outros valores compostos com facilidade. O exemplo 5.32 apresenta o comando *foreach* de PERL, o qual permite a iteração sobre os elementos de listas e vetores.

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

Exemplo 5. 32 - Comando *foreach* de PERL

Embora seja muito vantajoso, poucas LPs oferecem comandos que percorrem diretamente coleções de elementos. Uma vez que não é possível encontrar uma forma genérica para percorrer os diferentes tipos de coleções e, em particular, conhecer de antemão como essas coleções são implementadas, LPs optam por deixar para o implementador da coleção a responsabilidade de criar uma função iteradora para cumprir essa funcionalidade. Cabe destacar que LPs orientadas a objetos, como C++ e JAVA, oferecem objetos iteradores associados às classes de coleção de sua biblioteca padrão os quais proporcionam exatamente essa funcionalidade.

5.2.1.6 Chamadas de Procedimentos

Assim como chamadas de funções são expressões, chamadas de procedimentos são comandos. Chamadas de procedimento têm por objetivo atualizar variáveis. Isso é feito atualizando variáveis passadas pela lista de parâmetros ou alterando o valor de variáveis não locais.

A diferença mais importante de um procedimento para uma função é o fato do primeiro não retornar um valor. Dessa forma, boa parte do que foi discutido na seção sobre chamadas de função (seção 5.1.18) também se aplica às chamadas de procedimento. Tal como uma chamada de função, a chamada de um procedimento também é feita através da aplicação do nome do procedimento a um ou mais valores correspondentes aos parâmetros do procedimento. Tal como uma chamada de função, uma chamada de procedimento pode ter qualquer aridade e o seu formato pode ser posicional ou por palavras chave.

5.1.2.7 Desvios Incondicionais

Todos os comandos listados até aqui exibem um fluxo de controle do tipo entrada e saída única. Esse padrão é adequado para a maioria dos propósitos e incentiva a construção de programas estruturados, mais fáceis de se ler e de se manter. Contudo, em certas situações ele pode ser muito restritivo.

Existe consenso que comandos que permitem entrada única e saída múltipla podem ser bastante úteis. Por outro lado, comando com entradas múltiplas são verdadeiramente nocivos a programação estruturada e devem ser evitados, senão proibidos.

Para permitir um fluxo de controle com entradas e saídas múltiplas, aproximadamente todas LPs imperativas dispõem de comandos de desvio incondicional, tais como desvios irrestritos, escapes e exceções. Essas últimas não serão discutidas aqui pois serão apresentadas em um capítulo próprio (o capítulo 8).

5.2.1.7.1 Desvios Irrestritos

Esse comando transfere a execução do programa para qualquer ponto especificado através de um rótulo. O comando de desvio irrestrito (mais conhecido como *goto*) confere um grande poder e flexibilidade ao programador. De fato, todos os demais comandos de fluxo de controle podem ser construídos a partir do *goto* e de uma operação de seleção.

Por outro lado, esse comando oferece liberdade excessiva ao programador. Isso pode ser nocivo a boa qualidade de programação (DIJKSTRA, 1968), facilitando a ocorrência da chamada programação macarrônica. Ao contrário do que postula a programação estruturada (estabelece que a legibilidade de programas é superior quando a execução segue a ordem na qual os comandos aparecem), programas com *goto* tendem a não seguir ordem alguma. Por conta disso, alguns estudiosos de programação desejaram que o comando *goto* fosse banido das LPs. Esse desejo foi atendido em MODULA-2. Essa LP não possui o comando *goto* ou equivalente.

Contudo, em certas ocasiões, o ganho em eficiência que o *goto* pode proporcionar compensa uma possível perda de legibilidade [KNUTH, 1974]. Em particular, o uso mais comum do *goto* é para abandonar o processamento em um fluxo de controle aninhado, tal como no caso em que se deseja sair de duas repetições aninhadas ao mesmo tempo. O exemplo 5.33 usa o comando *goto* de C em uma situação como essa.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
```

```
        if ( a[i] == b[j] ) goto saida;
saida:
if (i < n) printf ("achou!!!");
else printf ("não achou!!!");
```

Exemplo 5. 33 - Uso de Comando *goto* em C

O exemplo 5.33 compara os elementos de dois vetores *a* e *b*. Ao encontrar um elemento comum, o fluxo de controle é transferido para a linha rotulada com o identificador *saida*. Nesse caso, o comando *goto* evita que seja necessário continuar as repetições até o final. Observe que o código escrito com *goto* pode ser reescrito sem ele, mas com o custo da inserção de uma variável adicional e da realização de alguns testes repetidos. O exemplo 5.34 mostra essa alternativa.

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

Exemplo 5. 34 - Código Equivalente em C sem o uso de *goto*

Existem outras situações nas quais o comando *goto* pode facilitar a programação. Um exemplo é na propagação de condições de erros de um subprograma mais interno para um mais externo. Esse recurso é muito utilizado por LPs que não oferecem mecanismos de tratamento de exceções, tal como PASCAL.

Um último comentário sobre desvios irrestritos diz respeito aos rótulos usados em associação ao comando *goto*. ADA requer que os identificadores de rótulos sejam envoltos entre << e >>. PASCAL só permite rótulos numéricos.

5.2.1.7.2 Escapes

Escapes são usados para permitir uma finalização diferente da usual em um comando, subprograma ou programa. Comandos de escapes são desvios incondicionais considerados estruturados, uma vez que só permitem a realização de desvios disciplinados no fluxo de controle do programa. Em contraste com os comandos de desvio irrestrito, eles não podem ser usados para criar ou entrar em repetições, nem para desviar o fluxo de controle para um local qualquer do programa.

O escape mais comum permite sair de comandos iterativos. C, C++ e JAVA possuem o comando de escape *break*. Esse comando também é

usado para sair de comandos condicionais de caminhos múltiplos. O exemplo 5.35 ilustra o uso do comando *break* em C.

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

Exemplo 5. 35 - O Comando *break* de C

O exemplo 5.35 realiza a mesma funcionalidade que os comandos iterativos do exemplo 5.28. Note a falta de necessidade de repetir os comandos *printf* e *scanf* para leitura de *n* e de duplicar a verificação do valor de *n*. Isso é possível porque o comando *break* possibilita colocar o ponto de parada (isto é, a saída) do comando iterativo em qualquer local do seu corpo.

De fato, a combinação de um comando de repetição infinita como o *for(;;)*, um comando simples de seleção como o *if* e um comando de escape como o *break* é suficiente para atender todas as demandas por comandos iterativos com número de repetições indefinido. Por exemplo, para substituir o comando com pré-teste basta colocar o ponto de parada imediatamente após o *for(;;)* e para substituir o comando com pós-teste basta colocar o ponto de parada imediatamente antes do final do corpo da repetição.

Outro tipo de escape existente em C, C++ e JAVA é o comando *continue*. Ele permite continuar o comando iterativo a partir do ponto onde foi colocado. Nesse caso, ao invés de finalizar o comando iterativo, o escape é usado para passar diretamente para o início da próxima iteração. O exemplo 5.36 mostra o uso do comando *continue* para somar dez números naturais.

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

Exemplo 5. 36 - O Comando *continue* de C

A associação de escapes com iterações rotuladas pode ser útil em situações nas quais se quer sair ou continuar uma repetição mais externa a partir de uma mais interna. Enquanto C, C++ e MODULA-2 não permitem escapes rotulados, ADA e JAVA oferecem este tipo de escape. O exemplo 5.37 mostra o uso de escapes rotulados em JAVA em um trecho de programa que identifica se existe um elemento comum presente em dois vetores ordenados.

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

Exemplo 5. 37 - Escapes Rotulados em JAVA

Outra categoria comum de escapes interrompe a execução de subprogramas ou programas. C possui utiliza o escape *return* para interromper a execução de um subprograma e a função da biblioteca padrão *exit* para interromper a execução de programas. O exemplo 5.38 ilustra o uso desses comandos em C.

```
void trata (int erro) {
  if (erro == 0) {
     printf ("nada a tratar!!!");
     return;
  }
  if (erro < 0) {
     printf ("erro grave – nada a fazer!!!");
     exit (1);
  }
  printf("erro tratado!!!");
}
```

Exemplo 5. 38 - Escapes de Subprograma e Programa em C

No exemplo 5.38, o escape *return* é usado para sair da função *trata* quando não ocorreu erro e o escape *exit* é usado para terminar o programa quando o erro é grave demais para ser tratado.

A combinação de comandos de escapes, escapes rotulados e mecanismo de tratamento de exceções tem possibilitado a JAVA não possuir comando de desvio irrestrito. O comando *goto* é uma palavra reservada em JAVA, porém não faz parte da linguagem atualmente. Isso significa que, em

princípio, os projetistas de JAVA não consideram que o comando *goto* seja realmente necessário. Contudo, eles deixam aberta a possibilidade de incorporar esse comando à linguagem, caso seja constatada sua necessidade para atender usos não contemplados por aquelas outras construções da LP.

5.3 Considerações Finais

Nesse capítulo foram discutidos os diversos tipos fundamentais de expressões e comandos existentes em LPS imperativas. Uma LP imperativa fica empobrecida quando não possui ou restringe arbitrariamente qualquer dos tipos de expressões e comandos apresentados aqui. Por outro lado, expressões e comandos adicionais podem muito bem não acrescentar nada ao poder expressivo da LP. Por exemplo, comandos de entrada e saída existentes em FORTRAN e COBOL poderiam ser substituídos por chamadas de procedimentos (tal como em PASCAL, C e ADA).

Pode-se observar uma certa interseção entre as funcionalidade de alguns tipos de expressões e comandos. Expressões e comandos condicionais, expressões com efeito colateral e atribuições, chamadas de funções e procedimentos têm uma correlação funcional. Por conta disso, algumas LPs optaram por suprimir todas as distinções entre comandos e expressões. São chamadas de LPs orientadas a expressões. Nessas LPs, a avaliação de uma expressão retorna um valor e pode ter como efeito colateral a atualização de uma variável. São exemplos de LPs orientadas a expressão ALGOL-68 e ML.

C também pode ser considerada uma LP orientada a expressões desde que se considere o retorno das operações de fluxo de controle como *void*. Assim, a atribuição $v = e$ é uma expressão que retorna o valor da expressão e junto com o efeito colateral de atribuir esse valor a v. Dessa maneira, operações como atribuição múltipla $u = v = e;$ se tornam natural.[5.6]

As principais vantagens das LPs orientadas a expressões são a obtenção de maior simplicidade, visto que se evita a duplicação entre comandos e expressões análogas, e uniformidade, visto que se elimina a separação das operações de controle de fluxo e atribuição em comandos e expressões. A principal desvantagem dessa abordagem é encorajar o uso de efeitos colaterais, o que frequentemente incentiva a má prática de programação.

[5.6] Algumas LPs orientadas a expressão retornam *0* ou *void* para a atribuição. Por exemplo, ML.

5.4 Exercicios

1. Um programa deve ler uma sequência de números inteiros e imprimí-los. O programa deve ser interrompido quando o número lido for zero. Implemente três versões desse programa em C usando respectivamente os comandos iterativos com pré-teste, com pós-teste e um comando de escape. Discuta as soluções apresentadas em termos de redigibilidade e eficiência, indicando a melhor solução apresentada.

2. Descreva o que ocorre em cada trecho que culmina com impressões no seguinte programa em C, justificando suas afirmações.

```
#include <stdio.h>
main () {
   int a, b, c;
   b = c = 10;
   a = b++ + b++;
   printf("%d\n",  a );
   printf("%d\n",  b );
   a = ++c + ++c;
   printf("%d\n",  a );
   printf("%d\n",  c );
   b = 10;
   a = b++ + b;
   printf("%d\n",  a );
   printf("%d\n",  b );
   a = 10;
   b = 5;
   if (a>b || ++b>5)
     printf("%d\n", b);
   a = 1;
   b = 5;
   if (a>b || ++b>5)
     printf("%d\n", b);
}
```

   Esse programa em C é portável? O que ocorreria se um programa equivalente (isto é, usando classe e com o comando apropriado de saída) fosse implementado em JAVA? Justifique todas as suas afirmações.

3. Faça um programa em C (sem usar os escapes *return* ou *exit*) para ler um número inteiro positivo e calcular a tabuada de multiplicação de 0 a 9 de todos os números positivos inferiores ao número lido. Sempre que o resultado de uma multiplicação for múltiplo de dez, o programa

deve perguntar ao usuário se ele deseja continuar com o cálculo da tabuada. Portanto, o programa deve se encerrar ao final do cálculo da tabuada ou quando o usuário responder que não deseja continuar o cálculo. A sua solução é a mais eficiente para esse problema? Porque? Como a solução mais eficiente seria implementada em JAVA?

4. Apresente um exemplo de expressão em C onde ocorre curto-circuito associado a efeito colateral. Analise o efeito que tal expressão pode produzir sobre a legibilidade de um programa.

5. O comando *goto* é empregado para realizar desvios incondicionais no fluxo de controle de programas. Com o advento das técnicas de programação estruturada, este comando foi muito criticado e, por muitas vezes, se sugeriu que linguagens de programação não deveriam incluí-lo entre seus comandos. C é uma linguagem que adota os princípios da programação estruturada. No entanto, C manteve o *goto* como um comando da linguagem. Qual a razão dessa decisão? Exemplifique, com um trecho de programa em C, uma situação na qual pode ser útil empregar o comando *goto*. Discuta como programadores da linguagem MODULA-2 e JAVA, que não incluem o *goto* entre seus comandos, lidam com esta situação.

6. Diga qual o valor das variáveis *a* e *n* após cada linha do seguinte trecho de código C. Justifique suas respostas.

```
n = 3;
a = - n ++;
a = - n + 1;
a = - n += 1;
```

7. Modifique o seguinte trecho de código para que ele realize a semântica sugerida pela sua disposição textual.

```
if ( x == 7 )
        if ( y == 11)
                z = 13;
else z = 17;
```

8. Veja como funciona o exemplo 5.37. Execute-o passo a passo considerando que *n* tem valor *8* e os vetores *a* e *b* foram definidos da seguinte maneira:

```
int[] a = {1, 3, 6, 9, 10, 12, 16, 18};
int[] b = {2, 4, 5, 7, 8, 10, 11, 15};
```

Reimplemente esse exemplo em C garantindo que ele funcione exatamente como em JAVA.

9. Implemente e teste o seguinte programa em C e descreva o que acontece. Justifique porque isso ocorre dessa maneira (isto é, apresente o racional da decisão tomada pelos projetistas ou implementadores dessa LP).

```
void f() {
  int i = 10;
  entra:
  i++;
}
main() {
  f();
  goto entra;
}
```

10. Analise o seguinte programa em C, identificando o que ele faz. Faça uma crítica ao estilo de programação utilizado.

```
main() {
  int i, j, k;
  k = 1;
  for (i = 0; i < 10; i++) {
        entra:
        j = 2 * i + 1;
        printf("i: %d, j: %d\n", i, j);
  }
  if (k) {
        k = 0;
        i = 7;
        goto entra;
  }
}
```

11. Algumas LPs (tal como, C) consideram a operação de atribuição como sendo uma espécie de expressão (isto é, a atribuição é uma operação que retorna um valor). Dê exemplos de situações nas quais isso pode ser vantajoso. Diga também quando essa característica pode ser danosa para a qualidade da programação. Justifique sua resposta.