2.3. UM EXEMPLO: INFIXO, POSFIXO E PREFIXO

DEFINIÇÕES BÁSICAS E EXEMPLOS

Esta seção examinará uma importante aplicação que ilustra os diferentes tipos de pilhas e as diversas operações e funções definidas a partir delas. O exemplo é, em si mesmo, um relevante tópico de ciência da computação.

Considere a soma de *A* mais *B.* Imaginamos a aplicação do ***operador*** "+" sobre os ***operandos*** *A* e *B,* e escrevemos a soma como *A + B.* Essa representação particular é chamada ***infixa.*** Existem duas notações alternativas para expressar a soma de *A* e *B* usando os símbolos *A, B* e +. São elas:

+ A B prefixa

A B + posfixa

Os prefixos "pre", "pos" e "in" referem-se à posição relativa do operador em relação aos dois operandos. Na notação prefixa, o operador precede os dois operandos; na notação posfixa, o operador é introduzido depois dos dois operandos e, na notação infixa, o operador aparece entre os dois operandos. Na realidade, as notações prefixa e posfixa não são tão incômodas de usar como possam parecer a princípio. Por exemplo, uma função em C para retornar a soma dos dois argumentos, *A e B, é* chamada por *add(A, B).* O operador *add* precede os operandos *A* e *B.*

Examinemos agora alguns exemplos adicionais. A avaliação da expressão *A + B* * C, conforme escrita em notação infixa, requer o conhecimento de qual das duas operações, + ou *, deve ser efetuada em primeiro lugar. No caso de + e *, "sabemos" que a multiplicação deve ser efetuada antes da adição (na ausência de parênteses que indiquem o contrário). Sendo assim, interpretamos *A + B * C* como *A + (B * C),* a menos que especificado de outra forma. Dizemos, então, que a multiplicação tem ***precedência*** sobre a adição. Suponha que queiramos reescrever *A + B * C* em notação posfixa. Aplicando as regras da precedência, converteremos primeiro a parte da expressão que é avaliada em primeiro lugar, ou seja a multiplicação. Fazendo essa conversão em estágios, obteremos:

A + (B * C)	parênteses para obter ênfase
A + (BC •)	converte a multiplicação
A (BC *) +	converte a adição
ABC * +	forma posfixa

As únicas regras a lembrar durante o processo de conversão é que as operações com a precedência mais alta são convertidas em primeiro lugar e que, depois de uma parte da expressão ter sido convertida para posfixa, ela deve ser tratada como um único operando. Examine o mesmo exemplo com a precedência de operadores invertida pela inserção deliberada de parênteses.

(A + B) * C	forma infixa
(AB +) * C	converte a adição
(AB +) C *	converte a multiplicação
AB + C*	forma posfixa

Nesse exemplo, a adição é convertida antes da multiplicação por causa dos parênteses. Ao passar de *(A + B) * C* para *(AB +) * C*, *A* e *B* são os operandos e + é o operador. Ao passar de *(AB +) * C* para (AB +)C *, (AB +) e *C* são os operandos e * é o operador. As regras para converter da forma infixa para a posfixa são simples, desde que você conheça as regras de precedência.

Consideramos cinco operações binárias: adição, subtração, multiplicação, divisão e exponenciação. As quatro primeiras estão disponíveis em C e são indicadas pelos conhecidos operadores +, -, * e /. A quinta operação, exponenciação, é representada pelo operador $. O valor da expressão *A $ B é A* elevado à potência de *B,* de maneira que 3 $ 2 é 9. Veja a seguir a ordem de precedência (da superior para a inferior) para esses operadores binários:

exponenciação
multiplicação/divisão
adição/subtração

Quando operadores sem parênteses e da mesma ordem de precedência são avaliados, pressupõe-se a ordem da esquerda para a direita, exceto no caso da exponenciação, em que a ordem é supostamente da direita para a esquerda. Sendo assim, *A + B + C* significa *(A + B) + C*, enquanto *A $ B $ C* significa *A $ (B* $ C). Usando parênteses, podemos ignorar a precedência padrão.

Apresentamos os seguintes exemplos adicionais de conversão da forma infixa para a posfixa. Procure entender cada um dos exemplos (e fazê-los por conta própria) antes de prosseguir com o restante desta seção.

Forma Infixa	Forma Posfixa
A + B	AB +
A + B - C	AB + C -
(A + B) * (C - D)	AB + CD - *
A $ B * C - D + E / F / (G + H)	A B $ C * D - E F / G H + / +
((A + B) * C - (D - E)) $ (F + G)	AB + C * DE - - FG + $
A - B / (C * D $ E)	ABCDE $ * / -

As regras de precedência para converter uma expressão da forma infixa para a prefixa são idênticas. A única mudança da conversão posfixa é que o operador é posicionado antes dos operandos, em vez de depois deles. Apresentamos as formas prefixas das expressões anteriores. Mais uma vez, você deve tentar fazer as transformações por conta própria.

Forma Infixa	Forma Prefixa
A + B	+ AB
A + B - C	- + ABC
(A + B) * (C - D)	* + AB - CD
A $ B * C - D + E / F / (G + H)	+ - * $ ABCD / / EF + GH
((A + B) * C - (D - E) $ (F + G)	$ - + ABC - DE + FG
A - B/(C * D $ E)	- A/B * C $ DE

Observe que a forma prefixa de uma expressão complexa não representa a imagem espelhada da forma posfixa, como podemos notar no segundo exemplo apresentado anteriormente, *A + B - C*. De agora em diante, consideraremos somente as transformações posfixas e deixaremos para o leitor, como exercício, a maior parte do trabalho envolvendo a forma prefixa.

Uma questão imediatamente óbvia sobre a forma posfixa de uma expressão é a ausência de parênteses. Examine as duas expressões, *A + (B * C)* e (A + *B) * C*. Embora os parênteses em uma das expressões sejam supérfluos [por convenção, *A + B*C=A + (B** C)], os parênteses na segunda expressão são necessários para evitar confusão com a primeira. As formas posfixas dessas expressões são:

Forma Infixa	Forma Posfixa
A + (B * C)	ABC* +
(A + B) * C	AB + C *

Não existem parênteses nas duas expressões transformadas. A ordem dos operadores nas expressões posfixas determina a verdadeira ordem das operações, ao avaliar a expressão, tornando desnecessário o uso de parênteses.

Ao passar da forma infixa para a posfixa, abrimos mão da possibilidade de observar rapidamente os operandos associados a um determinado operador. Entretanto, obtemos uma forma não-ambígua da expressão original sem o uso dos incômodos parênteses. Na verdade, a forma posfixa da expressão original poderia parecer mais simples, não fosse o fato de que ela parece difícil de avaliar. Por exemplo, como saberemos que, se $A = 3$, $B = 4$ e C = 5, nos exemplos anteriores, então 3 4 5* + equivalerá a 23 e 34 + 5* equivalerá a 35?

AVALIANDO UMA EXPRESSÃO POSFIXA

A resposta à pergunta anterior está no desenvolvimento de um algoritmo para avaliar expressões na forma posfixa. Cada operador numa string posfixa refere-se aos dois operandos anteriores na string. (Evidentemente, um desses operandos pode ser, ele mesmo, o resultado de aplicar um operador anterior.) Imagine que, cada vez que lemos um operando, nós o introduzimos numa pilha. Quando atingirmos um operador, seus operandos serão os dois primeiros elementos da pilha. Podemos, então, retirar esses dois elementos, efetuar a operação indicada sobre eles e introduzir o resultado na pilha para que ele fique disponível para uso como um operando do próximo operador. O seguinte algoritmo avalia uma expressão na forma posfixa, usando esse método:

```
opndstk = a pilha vazia;
/* verifica a primeira string lendo um */
/* elemento por vez para symb          */
while (nao terminar a entrada) {
   symb = proximo caractere de entrada;
```

```
   if (symb eh um operando)
      push(opndstk,  symb);
   else {
      /* symb eh um operador */
      opnd2 = pop(opndstk);
      opndl = pop(opndstk);
      value = resultado de aplicar symb a opnd1 e opnd2;
      push(opndstk,  value);
    }  /* fim else */
   }    /* fim while */
return(pop(opndstk));
```

Examinemos agora um exemplo. Imagine que sejamos solicitados a avaliar a seguinte expressão posfixa:

6 2 3 + - 3 8 2 / + * 2 $ 3 +

Apresentamos o conteúdo da pilha *opndstk* e as variáveis *symb*, *opndl*, *opnd2* e *value*, depois de cada iteração sucessiva da repetição. O topo de *opndstk* encontra-se à direita.

symb	opnd1	opnd2	value	opndstk
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
$	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Cada operando é introduzido na pilha de operandos quando encontrado. Portanto, o tamanho máximo da pilha é o número de operandos que aparecem na expressão de entrada. Entretanto, ao lidar com a maioria das expressões posfixas, o verdadeiro tamanho da pilha necessário será inferior a esse máximo teórico porque um operador remove operandos da pilha. No exemplo anterior, a pilha nunca continha mais que quatro elementos, apesar do fato de aparecerem oito operandos na expressão posfixa.

PROGRAMA PARA AVALIAR UMA EXPRESSÃO POSFIXA

Existem vários aspectos que precisamos considerar antes de realmente podermos escrever um programa para avaliar uma expressão em notação posfixa. Uma consideração básica, como acontece em todos os programas, é definir exatamente a forma e as restrições, se existir alguma, sobre a entrada. Em geral, o programador é apresentado a uma forma de entrada e é solicitado a elaborar um programa para acomodar os dados determinados. Por outro lado, estamos em posição de escolher a forma de nossa entrada. Isso nos permite construir um programa que não esteja sobrecarregado de problemas de transformações que camuflem o verdadeiro propósito da rotina. Se tivéssemos enfrentado dados numa forma desagradável e incômoda de trabalhar, poderíamos ter relegado as transformações a várias funções e usado a saída dessas funções como entrada em nossa rotina básica. No "mundo real", o reconhecimento e a transformação da entrada são questões importantes.

Imaginemos, nesse caso, que cada linha de entrada esteja na forma de uma string de dígitos e símbolos de operadores. Pressupomos que os operandos sejam dígitos não-negativos isolados, como 0, 1, 2, ..., 8, 9. Por exemplo, uma linha de entrada poderia conter 3 4 5* + nas cinco primeiras colunas seguidas por um caractere de final-de-linha ('\n'). Queremos escrever um programa que leia linhas de entrada nesse formato, enquanto existir alguma restante, e imprima para cada linha a string de entrada original e o resultado de avaliar a expressão.

Como os símbolos são lidos como caracteres, precisamos descobrir um método para converter os operandos de caracteres em números e os caracteres de operadores em operações. Por exemplo, precisamos de um método para converter o caractere '5' no número 5 e o caractere '+' na operação de adição.

A conversão de um caractere num inteiro pode ser facilmente manipulada em C. Se ***int*** *x* for um caractere de um só dígito em C, a expressão *x* - '0' resultará o valor numérico desse dígito. Para implementar a operação correspondente a um símbolo de operador, usamos a função *oper* que aceita a representação em caracteres de um operador e de dois operandos como parâmetros de entrada, e retorna o valor da expressão obtida pela aplicação do operador sobre os dois operandos. O corpo da função será apresentado mais adiante.

O corpo do programa principal poderia ser o seguinte. A constante *MAXCOLS* é o número máximo de colunas numa linha de entrada.

```
#define MAXCOLS 80
main()
{
   char expr[MAXCOLS];
   int position = 0;
   float eval();

   while ((expr[position++] = getchar ()) != '\n');
   expr[—position] = '\0';
   printf("%s%s", "a expressao posfixa original eh",
                                                    expr);
   printf("%f\n", eval(expr));
}  /* fim main */
```

Evidentemente, a parte principal do programa é a função *eval,* que vem a seguir. Essa função é apenas a implementação em C do algoritmo de avaliação, considerando o ambiente específico e o formato dos dados de entrada, e as saídas calculadas, *eval* chama uma função *isdigit* que determina se seu argumento é ou não um operando. A declaração para uma pilha, que aparece abaixo, é usada pela função *eval* que a acompanha, junto com as rotinas *pop* e *push,* que são chamadas por *eval.*

```
struct stack {
     int top;
     float items[MAXCOLS];
};

float eval(expr)
char expr [];
{
   int c, position;
   float opnd1, opnd2, value;
```

```
   float oper(), pop();
   struct stack opndstk;
  opndstk.top = -1
  for (position = 0 ; (c = expr[position]) != '\0';
                                                  position++)
      if (isdigit(c))
        /* operando-- converte a represent. em caractere */
        /* do digito em flutuante  e  introduz           */
        /*                    na pilha                   */
        push(&opndstk, (float) (c-'0'));
      else {
         /*                   operador                    */
         opnd2 = pop (&opndstk);
         opndl = pop (&opndstk);
         value = oper(c, opndl, opnd2);
         push (&opndstk, value);
      }  /* fim else */
  return (pop(&opndstk));
} /* fim eval */
```

Para completar, apresentamos *isdigit* e *oper.* A função *isdigit* verifica apenas se seu argumento é um dígito:

```
isdigit(symb)
char symb;
{
   return(symb >= '0'&& symb <= '9');
}
```

Essa função está disponível como uma macro predefinida na maioria dos sistemas C.

A função *oper* verifica se seu primeiro argumento é um operador válido e, se for, determina o resultado de sua operação nos dois argumentos seguintes. Para a exponenciação, pressupomos a existência de uma função *expon(opl, op2).*

```
float oper(symb, op1, op2)
int symb;
float opl, op2;
float expon();
{
   switch (symb) {
      case  '+' : return (opl + op2);
      case  '-' : return (opl - op2);
```

```
    case  '*' : return (op1 * op2);
    case  '/' : return (op1 / op2);
    case  '$' : return (expon(opl, op2));
   default   : printf("%s", "operação errada");
               exit(1);
  }   /* fim switch */
I* fim oper */
```

LIMITAÇÕES DO PROGRAMA

Antes de deixar o programa, precisamos ressaltar algumas de suas deficiências. Saber o que um programa não pode fazer é tão importante quanto saber o que ele pode fazer. É óbvio que a tentativa de usar um programa para solucionar um problema para o qual ele não foi elaborado leva ao caos. Pior ainda é a tentativa de solucionar um problema com um programa incorreto, só para fazer o programa gerar resultados incorretos sem o menor resquício de uma mensagem de erro. Nesses casos, o programador não receberá nenhuma indicação de que os resultados são incorretos e pode, portanto, fazer julgamentos falhos baseado nestes resultados. Por essa razão, é importante que o programador conheça as limitações do programa.

A principal falha desse programa é que ele não faz nada em termos de detecção e recuperação de erro. Se os dados em cada linha de entrada representarem uma expressão posfixa válida, o programa funcionará. Entretanto, suponha que uma linha de entrada tenha excesso de operadores ou operandos, ou que eles não estejam numa seqüência correta. Estes problemas poderiam resultar em alguém usando inadvertidamente o programa sobre uma expressão posfixa contendo números de dois dígitos, dando como resultado um número excessivo de operandos. Ou talvez o usuário do programa tenha a impressão de que o programa manipula números negativos e eles devam ser inseridos com o sinal de menos, o mesmo sinal usado para representar a subtração. Esses sinais de menos seriam tratados como operadores de subtração, resultando em excesso de operadores. Dependendo do tipo específico de erro, o computador pode tomar várias medidas (por exemplo, interromper a execução ou imprimir resultados errados).

Suponha que, na última instrução do programa, a pilha *opndstk* não esteja vazia. Não receberemos nenhuma mensagem de erro (porque não solicitamos nenhuma) e *eval* retornará um valor numérico para uma expres-

são que, provavelmente, foi incorretamente declarada em primeiro lugar. Suponha que uma das chamadas à rotina *pop* levante a condição de *underflow*. Como não usamos a rotina *popandtest* para retirar elementos da pilha, o programa será interrompido. Isso não parece razoável, uma vez que os dados incorretos em uma linha não devem impedir o processamento das linhas adicionais. Esses não são, de maneira alguma, os únicos problemas que podem surgir. Como exercício, talvez você possa escrever programas que acomodem entradas menos restritivas e alguns outros que detectem alguns dos erros já citados.

CONVERTENDO UMA EXPRESSÃO DA FORMA INFIXA PARA A POSFIXA

Até agora, apresentamos rotinas para avaliar uma expressão posfixa. Embora tenhamos discutido um método para transformar a forma infixa em posfixa, ainda não apresentamos um algoritmo para fazer isso. A partir de agora, concentraremos nossa atenção nessa tarefa. Assim que esse algoritmo for elaborado, poderemos ler uma expressão infixa e avaliá-la, convertendo-a primeiramente para posfixa e avaliando, em seguida, a expressão posfixa.

Em nossa discussão anterior, mencionamos que as expressões entre parênteses mais internos precisam ser primeiro convertidas em posfixas para que, então, sejam tratadas como operandos isolados. Dessa maneira, os parênteses podem ser sucessivamente eliminados até que a expressão inteira seja convertida. O último par de parênteses a ser aberto dentro de um grupo de parênteses encerra a primeira expressão dentro deste grupo a ser transformada. Esse comportamento do tipo "o último a entrar é o primeiro a sair" sugere imediatamente o uso de uma pilha.

Examine as duas expressões infixas, *A + B*Ce(A + B)*C,e* suas respectivas versões posfixas, *ABC * +* e *AB + C **. Em cada caso, a ordem dos operandos é idêntica à ordem dos operandos nas expressões infixas originais. Ao avaliar a primeira expressão, *A + B * C, o* primeiro operando, *A,* pode ser inserido de imediato na expressão posfixa. Evidentemente, o símbolo + não pode ser inserido antes de seu segundo operando, o qual ainda não foi verificado, ser inserido. Portanto, ele precisa ser armazenado para ser recuperado e inserido em sua posição correta. Quando o operando *B* for

verificado, ele será inserido imediatamente depois de *A.* Entretanto, agora os dois operandos foram verificados. O que impedirá que o símbolo + seja recuperado e inserido? Evidentemente, a resposta é o símbolo *, que vem a seguir e tem precedência sobre +. No caso da segunda expressão, o parêntese de fechamento indica que a operação + deve ser efetuada primeiro. Lembre-se de que, na forma posfixa, ao contrário da infixa, o operador que aparece antes na string será o primeiro aplicado.

Como a precedência desempenha um papel importante ao transformar a forma infixa em posfixa, vamos supor a existência de uma função *prcd(opl, op2),* onde *op1* e *op2* são caracteres representando operadores. Essa função retornará *TRUE,* se *opl* tiver precedência sobre *op2* quando *opl aparecer* à esquerda de *op2* numa expressão infixa, sem parênteses.*prcd(op1, op2)* retornará *FALSE,* caso contrário. Por exemplo, *prcd('*"+')* e *prcd('+','+')* são *TRUE,* enquanto prcd('+','*') é *FALSE.*

Apresentaremos agora um esboço de um algoritmo para converter uma string infixa sem parênteses numa string posfixa. Como não presumimos a existência de parênteses na string de entrada, o único controlador da seqüência na qual os operadores aparecerão na string posfixa é a precedência. (Os números de linhas que aparecem no algoritmo serão usados para referência futura.)

```
1   opstk = a pilha vazia;
2   while (nao terminar a entrada)   {
3      symb = proximo caractere de entrada;
4      if (symb for um operando)
          inclui symb na string posfixa
5      else {
6         while(lemptyfopstk) && prcd(stacktopfopstk),
                                                  symb))  {
7            topsymb = pop(opstk);
8            inclui topsymb na string posfixa;
          }  /* fim while */
9         push(opstk, symb);
       }  /* fim else */
     } /* fim while */
     /* saida de quaisquer operadores restantes */
10 while (!empty(opstk))  {
11    topsymb = pop(opstk);
12    inclui topsymb na string posfixa;
   }  /* fim while */
```

Simule o algoritmo com strings infixas, como "A * B + C * D" e "A + B * C $ D $ E" [onde '$' representa a exponenciação e *prcd* ('$', '$') equivale a *FALSE]* para se convencer de que ele está correto. Observe que, a cada

ponto da simulação, um operador na pilha tem uma precedência menor do que todos os operadores acima dele. Isso acontece porque a pilha vazia inicial satisfaz apenas essa condição, e um operador só é introduzido na pilha (linha 9) se o operador posicionado atualmente no topo da pilha tiver uma precedência menor que o operador sendo inserido.

Que alteração precisa ser feita nesse algoritmo para acomodar parênteses? A resposta é: uma alteração mínima. Quando um parêntese de abertura for lido, ele deverá ser introduzido na pilha. Isso pode ser feito, estabelecendo-se a convenção de *que prcd(op,'(')* é *FALSE* para todo símbolo de operador *op* diferente de um parêntese direito. Além disso, definimos *prcdÇÇ, op)* como *FALSE* para todo símbolo de operador *op.* [O caso de *op* = = ')' será discutido em breve.] Isso garantirá que um símbolo de operador aparecendo depois de um parêntese esquerdo será introduzido na pilha.

Quando um parêntese de fechamento for lido, todos os operadores até o primeiro parêntese de abertura deverão ser retirados da pilha para a string posfixa. Isso pode ser feito definindo-se *prcd(op,* ')') como *TRUE* para todos os operadores *op* diferentes de um parêntese esquerdo. Quando esses operadores forem removidos da pilha e o parêntese de abertura for descoberto, uma ação especial deve ser tomada. O parêntese de abertura deve ser removido da pilha e descartado, juntamente com o parêntese de fechamento, em vez de colocado na string posfixa ou na pilha. Vamos definir *prcdÇÇ,')')* com *FALSE.* Isto garantirá que, ao alcançar um parêntese de fechamento, a repetição começando na linha 6 será pulada para que o parêntese de abertura não seja inserido na string posfixa. A execução, portanto, prossegue até a linha 9. Entretanto, como o parêntese de fechamento não deve ser introduzido na pilha, a linha 9 é substituída pela instrução:

```
9     if (empty(opstk) | | symb != ')')
        push(opstk, symb);
      else /* remove o abre-parentese e descarta-o */
        topsymb = pop(opstk);
```

Com as convenções anteriores para a função *prcd* e a revisão da linha 9, o algoritmo pode ser usado para converter qualquer string infixa em posfixa. Resumimos as regras de precedência para parênteses:

```
prcd('(' ,op) = FALSE        para qualquer operador op
prcd(op, '(') = FALSE        p/ qq operador op difer. de ' ) '
prcd(op, ' (') = TRUE        p/ qq operador op difer. de '('
prcd('(' ,op) = não-definido   p/ qq operador op (uma tentativa
                             de comparar os dois indica um
                             erro)•
```

Ilustraremos esse algoritmo em alguns exemplos:

Exemplo 1: *A+B* C*

O conteúdo de *symb,* a string posfixa, e *opstk* apresentado depois da verificação de cada símbolo, *opstk* aparece com seu topo à direita.

	symb	string posfixa	opstk
1	*A*	A	
2	**+**	A	**+**
3	*B*	*AB*	**+**
4	*	AB	**+ ***
5	*C*	ABC	**+ ***
6		*ABC**	**+**
7		*ABC** +	

As linhas 1, 3 e 5 correspondem à verificação de um operando; portanto, o símbolo *(symb)* é imediatamente colocado na string posfixa. Na linha 2, um operador é verificado e a pilha é considerada vazia; o operador é, portanto, colocado na pilha. Na linha 4, a precedência do novo símbolo (*) é maior que a do símbolo posicionado no topo da pilha (+); portanto, o novo símbolo é colocado na pilha. Nos passos 6 e 7, a string de entrada está vazia, e a pilha é, portanto, esvaziada e seu conteúdo colocado na string posfixa.

Exemplo 2: *(A + B) * C*

symb	string posfixa	opstk
(		(
A	A	(
+	A	**(+**
B	AB	**(+**
)	AB +	
*	*AB* +	*****
C	*AB +C*	*****
	*AB +C**	

Neste exemplo, quando o parêntese direito é encontrado, a pilha é esvaziada até que um parêntese esquerdo seja encontrado, em cujo ponto ambos os parênteses são descartados. Usando parênteses para impor uma ordem de precedência diferente da padrão, a ordem de aparição dos operadores na string posfixa é diferente da seqüência apresentada no exemplo 1.

Exemplo 3: ((A - *(B* + C) * *D)* $ *E* + *F)*

symb	string posfixa	opstk
(		(
(		((
A	*A*	((
	A	((-
(	*A*	((-(
B	*AB*	((-(
+	*AB*	**((-(+**
C	*ABC*	**((-(+**
)	*ABC+*	**((-**
)	*ABC+-*	**(**
*	*ABC* + -	**(***
D	*ABC* + -D	**(***
)	*ABC* + *-D**	
$	*ABC* + *-D**	**$**
(	*ABC* + *-D**	**$(**
E	*ABC* + *-D** *E*	**$(**
+	*ABC* +*-D* * E	$(+
F	*ABC* + *-D** *EF*	**$(+**
)	*ABC* + *-D** *F* +	**$**
	ABC + *-D*EF+$*	

Por que o algoritmo de conversão parece tão complicado, enquanto o algoritmo de avaliação parece simples? A resposta é que o primeiro converte a partir de uma ordem de precedência (controlada pela função *prcd* e pela

presença de parênteses) para a ordem natural (isto é, a operação a ser executada primeiro aparece primeiro). Por causa das diversas combinações de elementos no topo da pilha (se não estiver vazia) e do possível símbolo a ser introduzido, é necessário um grande número de instruções para garantir que toda possibilidade seja coberta. No último algoritmo, por outro lado, os operadores aparecem exatamente na ordem em que serão executados. Por essa razão, os operandos podem ser empilhados até que um operador seja encontrado, em cujo ponto a operação é efetuada imediatamente.

A motivação por trás do algoritmo de conversão é o desejo de dar saída aos operadores, na ordem em que eles devem ser executados. Ao solucionar este problema manualmente, poderíamos seguir vagas instruções que nos exigissem converter de dentro para fora. Isso funciona bem para os humanos solucionando um problema com lápis e papel (caso eles não se confundam ou cometam um erro). Entretanto, um programa ou um algoritmo precisa ser mais exato em suas instruções. Não podemos ter certeza de ter alcançado os parênteses mais internos ou o operador com a maior precedência, até que símbolos adicionais tenham sido verificados. No momento, precisamos voltar a alguns pontos anteriores.

Em vez de retroceder continuamente, fazemos uso da pilha para "lembrar" os operadores encontrados anteriormente. Se um operador introduzido tiver uma precedência maior que o posicionado no topo da pilha, esse novo operador será introduzido na pilha. Isso significa que, quando todos os elementos da pilha forem finalmente removidos, esse novo operador precederá o topo anterior na string posfixa (o que estará correto, porque ele tem maior precedência). Por outro lado, se a precedência do novo operador for menor que a do operador posicionado no topo da pilha, o operador no topo da pilha deverá ser executado em primeiro lugar. Portanto, o topo da pilha é removido e o símbolo introduzido é comparado ao novo topo, e assim por diante. Os parênteses na string de entrada ignoram a ordem das operações. Sendo assim, quando um parêntese esquerdo for verificado, ele será colocado na pilha. Quando seu parêntese direito associado for encontrado, todos os operadores entre os dois parênteses serão colocados na string de saída, porque eles deverão ser executados antes de quaisquer operadores que apareçam depois dos parênteses.

PROGRAMA PARA CONVERTER UMA EXPRESSÃO DA FORMA INFIXA NA FORMA POSFIXA

Precisamos fazer duas coisas antes de começar a escrever um programa. A primeira é definir exatamente o formato da entrada e da saída. A segunda é construir, ou pelo menos definir, as rotinas das quais a rotina principal dependerá. Pressupomos que a entrada consiste em strings de caracteres, uma string por linha de entrada. O final de cada string é indicado pela presença de um caractere de final de linha C\n'). Para simplificar, presumimos que todos os operandos sejam letras ou dígitos de um só caractere. Todos os operadores e parênteses são representados por si mesmos, e '$' representa a exponenciação. A saída é uma string de caracteres. Essas convenções tornam a saída do processo de conversão adequada para o processo de avaliação, desde que todos os operandos de um só caractere na string infixa inicial sejam dígitos.

Ao transformar o algoritmo de conversão num programa, fazemos uso de várias rotinas. Entre elas, *empty*, *pop*, *push* e *popandtest*, todas adequadamente modificadas de modo que os elementos na pilha sejam caracteres. Fazemos uso também de uma função, *isoperand,* que retorna *TRUE* se seu argumento for um operando, e *FALSE,* caso contrário. Essa função simples será deixada para o leitor.

De modo semelhante, a função *prcd* será deixada como exercício para o leitor. Ela aceita dois símbolos de operadores de um único caractere como argumentos e retorna *TRUE* se o primeiro tiver precedência sobre o segundo quando aparecer à esquerda do segundo numa string infixa, e *FALSE,* caso contrário. Evidentemente, a função deverá incorporar as convenções de parênteses apresentadas anteriormente.

Assim que estas funções auxiliares forem escritas, poderemos escrever a. função de conversão, *postfix,* e um programa que a chamará. O programa lerá uma linha contendo uma expressão na forma infixa, chamará a rotina *postfix* e imprimirá a string posfixa. Veja a seguir o corpo da rotina principal:

```
#define MAXCOLS 80
main()
{
   char infix[MAXCOLS];
```

```
   char postr[MAXCOLS];
   int pos = 0;

   while ((infix[pos++] = getchar()) != '\n');
   infix[—pos] = '\0';
   printf("%s%s", "a expressão infixa original eh",infix);
   postfix(infix, postr);
   printf("%s\n", postr);
} /* fim main */
```

A seguir, você encontrará a declaração para a pilha de operadores e a rotina *postfix:*

```
struct stack {
     int top;
     char items[MAXCOLS];
};

postfix(infix, postr)
char infix[];
char postr[];
{
   int position, und;
   int outpos = 0;
   char topsymb = '+';
   char symb;
   struct stack opstk;
   opstk.top = -1       /* a pilha vazia */

   for (position=0; (symb = infix[position]) != '\0';
                                                 position++)
      if (isoperand(symb))
        postr[outpos++1] = symb;
      else {
         popandtest (&opstk, &topsymb, &und);
         while (!und && prdc(topsymb, symb)) {
            postr[outpos++] = topsymb;
            popandtest (&copstk, &topsymb, &und);
         }  /* fim while */
         if (!und)
            push (&copstk, top symb);
         if (und || (symb != ')'))
            push(&opstk, symb);
         else
            topsymb = pop (&opstk);
```

```
    }   /* fim else */
  while (!empty(&opstk))
     postr[outpos++] = pop(&opstk);
  postr[outpos] = '\0';
 return;
} /* fim postfix */
```

O programa apresenta uma grande deficiência porque ele não verifica se a string de entrada é uma expressão infixa válida. Na verdade, seria instrutivo para você examinar a operação desse programa quando apresentado com uma string posfixa válida como entrada. Como exercício, solicitamos que você escreva um programa que verifique se uma string de entrada é uma expressão infixa válida ou não.

Agora, podemos escrever um programa para ler uma string infixa e calcular seu valor numérico. Se a string original consistir em operadores de um só dígito sem operandos que sejam letras, o seguinte programa lerá a string original e imprimirá seu valor.

```
#define MAXCOLS 80
main()
{
   char instring[MAXCOLS], postring[MAXCOLS];
   int position = 0;
   float eval ();

   while((instring[position++] = getchar()) !='\n');
   instring[—position] = '\0';
   printf("%s%s", "a expressão infixa eh", instring);
   postfix(instring, postring);
   printf("%s7f\n", "valor e", eval(postring));
}  /* fim main */
```

São necessárias duas versões diferentes das rotinas de manipulação de pilha *ipop, push* e outras) porque *postfix* usa uma pilha de operadores de caracteres (isto é, *opstk),* enquanto *eval* usa uma pilha de operandos flutuantes (isto é, *opndstk).* Evidentemente, é possível usar uma única pilha contendo tanto reais como caracteres definindo uma união, conforme descrito anteriormente na Seção 1.3.

Dedicamos mais atenção, nesta seção, às transformações envolvendo expressões posfixas. Um algoritmo para converter uma expressão infixa em posfixa verifica caracteres da esquerda para a direita, empilhando e desempilhando, conforme a necessidade. Se fosse necessário converter da forma

infixa para a prefixa, a string infixa poderia ser verificada da direita para a esquerda e os símbolos corretos poderiam ser inseridos na string prefixa da direita para a esquerda. Como a maioria das expressões algébricas é lida da esquerda para a direita, a forma posfixa representa uma opção mais natural.

Os programas anteriores são simples indicativos do tipo de rotinas que se pode escrever para manipular e avaliar expressões posfixas. De modo algum, eles são abrangentes ou exclusivos. Existem diversas variações das rotinas anteriores igualmente aceitáveis. Alguns dos primeiros compiladores de linguagem de alto nível usavam realmente rotinas, como *eval* e *postfix,* para manipular expressões algébricas. Desde aquela época, foram desenvolvidas técnicas mais sofisticadas para manipular tais problemas.

EXERCÍCIOS

2.3.1. Transforme cada uma das seguintes expressões em prefixas e posfixas:

a. *A + B - C*

b. (A + B)*(C -*D)$E** F*

e. (A + *B)*(C $(D-E) + F)-G*

d. (A + (((B - *C)*(D -E)+F) I G)$(H - J)*

2.3.2. Transforme cada uma das seguintes expressões prefixas em infixas:

a. + - *ABC*

b. +*A-BC*

e. + + A - * $ *BCD/ + EF * GHI*

d. + - $ *ABC * D ** EFG*

2.3.3. Transforme cada uma das seguintes expressões posfixas em infixas:

a. *AB + C* -

b. *ABC* + -

c. *AB -C + DEF - + $*

d. *ABCDE - + $ * EF * -*

2.3.4. Aplique o algoritmo de avaliação apresentado neste capítulo para avaliar as seguintes expressões posfixas. Pressuponha que *A* = 1, *B* = 2, *C* = 3.

a. *AB + C - BA + C $ -*

b. *ABC + * CBA - + **

2.3.5. Modifique a rotina *eval* de modo a aceitar como entrada uma string de caracteres de operadores e operandos representando uma expressão posfixa e criar a forma infixa totalmente com parênteses. Por exemplo, *AB +* seria transformada em *(A + B)* e *AB + C -* seria transformada em ((A + *B)* - C).

2.3.6. Escreva um único programa combinando os recursos de *eval* e *postfix* para avaliar uma string infixa. Use duas pilhas, uma para operandos e outra para operadores. Não converta primeiramente a string infixa em posfixa e, em seguida, avalie a string posfixa, mas, em vez disso, avalie no decorrer da operação.

2.3.7. Escreva uma rotina *prefix* para aceitar uma string infixa e criar a forma prefixa dessa string, presumindo que a string seja lida da direita para a esquerda e a string prefixa seja criada da direita para a esquerda.

2.3.8. Escreva um programa em C para converter:

a. uma string prefixa em posfixa

b. uma string posfixa em prefixa

e. uma string prefixa em infixa

d. uma string posfixa em infixa

2.3.9. Escreva uma rotina em C, *reduce,* que aceite uma string infixa e forme uma string infixa equivalente com todos os parênteses supérfluos removidos. Isso pode ser feito sem usar uma pilha?

2.3.10. Imagine uma máquina que possui um único registrador e seis instruções.

LD	***A***	coloca o operando A no registrador
ST	***A***	coloca o conteúdo do registrador na variável A
AD	***A***	soma o conteúdo da variável A ao registrador
SB	***A***	subtrai o conteúdo da variável A do registrador
ML	***A***	multiplica o conteúdo do registrador pela variável A
DV	***A***	divide o conteúdo do registrador pela variável A

Escreva um programa que aceite uma expressão posfixa contendo operandos de uma única letra e os operadores +, -, * e /, imprima uma seqüência de instruções para avaliar a expressão e deixe o resultado no registrador. Use variáveis da forma *TEMPn* como variáveis temporárias. Por exemplo, usar a expressão posfixa *ABC * + DE -I* deverá imprimir o seguinte:

```
LD      B
ML      C
ST      TEMP1
LD      A
AD      TEMP1
ST      TEMP2
LD      D
SB      E
ST      TEMP3
LD      TEMP2
DV      TEMP3
ST      TEMP4
```