EXPRESSÕES

Será mostrada agora uma interessante aplicação prática para a estrutura de dados pilha, implementada no Capítulo 2, que ilustra conceitos usados na construção de compiladores.

3.1 Fundamentos

Neste capítulo, usaremos o tipo `Pilha` para criar um programa que manipula *expressões* compostas por:

- *operandos*, que são constantes numéricas;
- *operadores*, que são operações aritméticas binárias (+, –, * e /);
- *delimitadores*, que são os parênteses de abertura e de fechamento.

Para facilitar a compreensão dos algoritmos, inicialmente, vamos supor que os operandos são números inteiros compostos por um *único* dígito.

3.1.1 Forma infixa

Normalmente, expressões aritméticas são escritas na *forma infixa*, isto é, com os operadores posicionados *entre* seus operandos. Por exemplo, `2*3+8/4` e `5*(7-3)`.

A ordem em que as operações são efetuadas numa expressão infixa depende de suas prioridades relativas. Por convenção, * e / têm prio-

ridade sobre + e −. Além disso, operadores de mesma prioridade devem ser efetuados na ordem em que eles aparecem na expressão. Por exemplo, ao avaliar a expressão `2*3+8/4`, primeiro efetuamos a multiplicação, depois a divisão e, por último, a soma.

Na forma infixa, parênteses servem para mudar a prioridade dos operadores. Por exemplo, na expressão `5*(7-3)`, os parênteses indicam que a subtração deve ser efetuada antes da multiplicação, mesmo tendo essa última uma prioridade maior. Quando os parênteses numa expressão infixa não mudam a prioridade dos operadores, podemos omiti-los. Assim, por exemplo, a expressão `(3*6)/4` pode ser escrita, de forma equivalente, como `3*6/4`.

3.1.2 Forma posfixa

Há dois fatores que dificultam a avaliação de uma expressão infixa: a *existência de prioridades*, que impede que as operações sejam efetuadas na ordem em que elas aparecem na expressão, e a *existência de parênteses*, que altera as prioridades relativas dos operadores usados na expressão.

Para resolver esse problema, o lógico polonês *Jan Lukasiewicz* propôs uma nova forma de escrever expressões chamada *forma posfixa* ou *notação polonesa reversa*. Nessa forma, os operadores são colocados *após* seus operandos. Assim, por exemplo, a conversão da expressão `2*3+8/4` para posfixa resulta em `23*84/+`.

Para converter a forma infixa em posfixa basta parentesiar completamente a expressão infixa, respeitando as prioridades dos operadores, e depois reescrever a expressão, descartando os parênteses e movendo os operadores para a posição ocupada por seus parênteses de fechamento, como indicado na Figura 3.1.

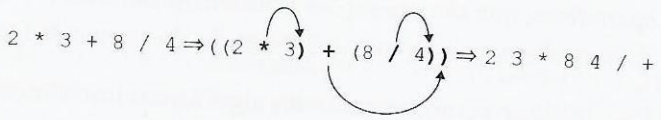


Figura 3.1 | Ideia para a conversão de expressão infixa em posfixa.

Como vemos na Figura 3.2, à medida que a expressão infixa é percorrida, os operandos encontrados são imediatamente copiados para a expressão posfixa; os operadores, porém, devem aguardar até que seus respectivos parênteses de fechamento sejam encontrados. Usando uma pilha como local de espera para os operadores, podemos obter o efeito desejado. A Figura 3.2 ilustra esse processo, indicando a ação tomada para cada elemento encontrado na expressão infixa.

Elemento	Ação	Pilha	Posfixa
(	descartar (	[]	""
(	descartar (	[]	""
2	anexar 2 à posfixa	[]	"2"
*	empilhar *	[*]	"2"
3	anexar 3 à posfixa	[*]	"23"
)	desempilhar * e anexar à posfixa	[]	"23*"
+	empilhar +	[+]	"23*"
(	descartar (	[+]	"23*"
8	anexar 8 à posfixa	[+]	"23*8"
/	empilhar /	[+,/]	"23*8"
4	anexar 4 à posfixa	[+,/]	"23*84"
)	desempilhar / e anexar à posfixa	[+]	"23*84/"
)	desempilhar + e anexar à posfixa	[]	"23*84/+"

Figura 3.2 | Conversão de infixa completamente parentesiada em posfixa.

A vantagem dessa conversão é que o valor de uma expressão posfixa pode ser facilmente obtido com uma pilha. Basta percorrer a expressão, da esquerda para a direita: quando um *operando* é encontrado, seu valor é empilhado; quando um *operador* é encontrado, dois valores são desempilhados e o resultado da operação feita com eles é empilhado. No final, o valor da expressão estará no topo da pilha. A Figura 3.3 ilustra esse processo, para a expressão posfixa `23*84/+`.

Elemento	Ação	Pilha
2	empilhar 2	[2]
3	empilhar 3	[2,3]
*	desempilhar 3 e 2 e empilhar 2*3	[6]
8	empilhar 8	[6,8]
4	empilhar 4	[6,8,4]
/	desempilhar 4 e 8 e empilhar 8/4	[6,2]
+	desempilhar 2 e 6 e empilhar 6+2	[8]

Figura 3.3 | Avaliação de expressão posfixa.

Como se pode observar na Figura 3.3, a ordem dos operadores na expressão posfixa corresponde à ordem em que eles devem ser avaliados. Isso torna o uso de parênteses desnecessário e simplifica o processo de avaliação.

3.2 Conversão de infixa parentesiada em posfixa

Para facilitar a compreensão da conversão de infixa em posfixa, vamos supor que a expressão infixa está *completamente parentesiada* (isto é, tem todos os parênteses possíveis). Assim, as prioridades dos operadores não serão levadas em conta.

A conversão de uma expressão infixa completamente parentesiada `e` em uma expressão posfixa `s` é feita pelo seguinte algoritmo:

- Inicie com uma pilha de caracteres `P` e uma expressão posfixa `s` vazias.
- Para cada elemento da expressão infixa `e`, da esquerda para a direita, faça:
  - Se for um *parêntese de abertura*, descarte-o.
  - Se for um *operando*, anexe-o à expressão posfixa `s`.
  - Se for um *operador*, insira-o na pilha `P`.
  - Se for um *parêntese de fechamento*, remova um item de `P` e anexe-o a `s`.

A função `posfixa()`, na Figura 3.4, implementa esse algoritmo. Ela recebe uma cadeia representando uma expressão infixa completamente parentesiada e devolve outra cadeia representando a expressão posfixa correspondente. A função `isdigit()`, declarada em `ctype.h`, verifica se um caractere é um dígito entre `'0'` e `'9'`. A função `strchr()`, declarada em `string.h`, é usada para verificar se uma cadeia contém um caractere específico (ela devolve o endereço da primeira ocorrência desse caractere na cadeia, ou `NULL`, se ele não for encontrado nela).

Como não é possível devolver um vetor como resposta de uma função em C, a função `posfixa()` devolve o endereço da variável local `s`. Para garantir que essa variável não seja destruída após a execução da função, ela é declarada com a classe de armazenamento `static` (isto é, com escopo local e duração global).

```
// posfixa.c - converte infixa completamente parentesiada em posfixa

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include "../ed/pilha.h"

char *posfixa(char *e) {
   static char s[256];
   int j=0;
   Pilha P = pilha(256);
   for(int i=0; e[i]; i++)
      if( isdigit(e[i]) ) s[j++] = e[i];
      else if( strchr("+*-/",e[i]) ) empilha(e[i],P);
      else if( e[i]==')' ) s[j++] = desempilha(P);
   s[j]='\0';
   destroip(&P);
   return s;
}

int main(void) {
   char e[256];
   printf("Infixa? ");
   gets(e);
   printf("Posfixa: %s\n\n", posfixa(e));
   return 0;
}
```

Figura 3.4 | Programa que converte infixa completamente parentesiada em posfixa.

3.2.1 Conversão de infixa em posfixa

Para converter em posfixa uma expressão infixa que *não* está completamente parentesiada, devemos observar que a ordem dos operandos na expressão infixa não muda na expressão posfixa; mas a ordem dos operadores muda (pois eles devem ser colocados após seus operandos).

Em uma expressão posfixa, as operações são efetuadas na ordem em que elas aparecem e, portanto, a posição de um operador na forma posfixa é definida pela sua prioridade na forma infixa. Operadores de maior prioridade na forma infixa devem aparecer primeiro na forma posfixa. Logo, um operador encontrado na forma infixa deve aguardar na pilha, até que apareça um operador com prioridade *menor ou igual* à dele (só então ele poderá ser movido para a forma posfixa).

Ademais, parênteses na forma infixa mudam as prioridades dos operadores. Por exemplo, os parênteses em `2*(9+1-5)` aumentam a prioridade da adição e da subtração em relação à multiplicação. Assim, um parêntese de abertura encontrado na forma infixa deve ser imediatamente empilhado, impedindo que outros operadores na pilha sejam avaliados antes daqueles na subexpressão que ele delimita. Porém, uma vez empilhado, a prioridade do parêntese de abertura torna-se mínima, aumentando a prioridade dos operadores na subexpressão por ele delimitada. A prioridade dos operadores é dada pela função na Figura 3.5.

```
int prio(char o) {
   switch( o ) {
      case '(': return 0;
      case '+':
      case '-': return 1;
      case '*':
      case '/': return 2;
   }
   return -1; // operador invalido!
}
```

Figura 3.5 | Função que informa a prioridade de um operador.

A conversão de uma expressão infixa `e` em uma expressão posfixa `s` é feita do seguinte modo:

- Inicie com uma pilha de caracteres `P` e uma expressão posfixa `s` vazias.
- Para cada elemento da expressão infixa `e`, da esquerda para a direita, faça:
  - Se for um *parêntese de abertura*, empilhe-o em `P`.
  - Se for um *operando*, anexe-o à expressão posfixa `s`.
  - Se for um *operador*, enquanto houver no topo da pilha `P` outro operador com maior ou igual prioridade, desempilhe esse operador e anexe-o a `s`; depois, empilhe em `P` o operador recém-encontrado na expressão `e`.

- Se for um *parêntese de fechamento*, remova um operador da pilha `P` e anexe-o a `s`, até que um parêntese de abertura apareça no topo na pilha. No final, desempilhe-lhe esse parêntese e descarte-o.

- Depois de percorrer completamente a expressão infixa `e`, esvazie a pilha, anexando à expressão posfixa `s` cada um dos operadores desempilhados.

Esse algoritmo é implementado na Figura 3.6 e seu funcionamento (para converter a expressão infixa `2*(7+3*5)-4`) é ilustrado na Figura 3.7.

```
char *posfixa(char *e) {
   static char s[256];
   int j=0;
   Pilha P = pilha(256);
   for(int i=0; e[i]; i++)
      if( e[i]=='(' ) empilha('(',P);
      else if( isdigit(e[i]) ) s[j++] = e[i];
      else if( strchr("+-/*",e[i]) ) {
         while( !vaziap(P) && prio(topo(P))>=prio(e[i]) )
            s[j++] = desempilha(P);
         empilha(e[i],P);
      }
      else if( e[i] == ')' ) {
         while( topo(P)!='(' )
           s[j++] = desempilha(P);
         desempilha(P);
      }
   while( !vaziap(P) )
      s[j++] = desempilha(P);
   s[j] = '\0';
   destroip(&P);
   return s;
}
```

Figura 3.6 | Função para conversão de infixa em posfixa.

Elemento de E	Ação correspondente	Pilha P	Posfixa S
2	anexar 2 à posfixa	[]	"2"
*	empilhar *	[*]	"2"
(	empilhar (	[*, (]	"2"
7	anexar 7 à posfixa	[*, (]	"27"
+	empilhar +	[*, (, +]	"27"
3	anexar 3 à posfixa	[*, (, +]	"273"
*	empilhar *	[*, (, +, *]	"273"
5	anexar 5 à posfixa	[*, (, +, *]	"2735"
)	desempilhar * e + e anexar à posfixa	[*, (]	"2735*+"
	desempihar (e descartar	[*]	"2735*+"
-	desempilhar * e anexar à posfixa	[]	"2735*+*"
	empilhar -	[-]	"2735*+*"
4	anexar 4 à posfixa	[-]	"2735*+*4"
fim	esvaziar e anexar - à posfixa	[]	"2735*+*4-"

Figura 3.7 | Conversão de infixa em posfixa.

3.3 Avaliação da forma posfixa

O valor de uma expressão posfixa `e` pode ser calculado pelo seguinte algoritmo:

- Inicie com uma pilha de inteiros `P` vazia.
- Para cada elemento da expressão `e`, da esquerda para a direita, faça:
  - Se for um *operando*, empilhe em `P` o seu valor numérico.
  - Se for um *operador*, desempilhe de `P` dois valores, aplique o operador a esses valores e empilhe em `P` o resultado obtido.
- No final, devolva como resultado o valor existente do topo de `P`.

Esse algoritmo é implementado pela função na Figura 3.8. Nessa função, a expressão `e[i]-'0'` converte um dígito em um número inteiro correspondente, usando subtração de códigos ASCII (*e.g.*, `'2'-'0'` equivale a `50-48`, que vale `2`). As variáveis `x` e `y` servem para ordenar os operandos corretamente num cálculo. Por exemplo, quando o operador `/` é encontrado na expressão posfixa `"862/-"`, os valores `2` e `6` são desempilhados, nessa ordem; porém, a conta a ser feita é `6/2` e não `2/6`. Usando `x` e `y`, a função pode ordenar esses operandos corretamente.

```
int valor(char *e) {
   Pilha P = pilha(256);
   for(int i=0; e[i]; i++)
      if( isdigit(e[i]) )
         empilha(e[i]-'0',P);
      else {
         int y = desempilha(P);
         int x = desempilha(P);
         switch( e[i] ) {
            case '+': empilha(x+y,P); break;
            case '-': empilha(x-y,P); break;
            case '*': empilha(x*y,P); break;
            case '/': empilha(x/y,P); break;
         }
      }
   int z = desempilha(P);
   destroip(&P);
   return z;
}
```

Figura 3.8 | Função para avaliação da forma posfixa.

3.4 Expressões com números reais de vários dígitos

Até agora, consideramos que os operandos nas expressões são inteiros de *um único* dígito. Nessa seção, mostraremos como adaptar os algoritmos apresentados para que eles funcionem com operandos reais compostos por vários dígitos.

3.4.1 Função de conversão

Numa expressão infixa, operandos são separados por operadores. Então, mesmo quando eles têm vários dígitos, é fácil identificá-los. Por exemplo, na expressão `"2*34+1"`, os operandos são `2`, `34` e `1`. Porém, na expressão posfixa `"234*1+"`, não há como saber se os operandos da multiplicação são os números `2` e `34` ou `23` e `4`.

Para eliminar essa ambiguidade, basta incluir um espaço na forma posfixa, sempre que um operador for encontrado na forma infixa. Assim, por exemplo, a conversão de `"2*34+1"` deve resultar em `"2 34 *1+"`.

Além disso, para que as expressões possam conter números reais, além dos dígitos, também devemos considerar a possibilidade de os operandos terem um ponto decimal. Assim, por exemplo, a expressão infixa `"2*34+1.79/5"` deve ser convertida em `"2 34 *1.79 5/+"`.

Uma versão do algoritmo de conversão de infixa em posfixa, com essas modificações, é apresentada na Figura 3.9.

```
char *posfixa(char *e) {
   static char s[256];
   Pilha P = pilha(256);
   int j=0;
   for(int i=0; e[i]; i++)
      if( e[i]=='(' ) empilha('(',P);
      else if( isdigit(e[i]) || e[i]=='.' ) s[j++] = e[i];
      else if( strchr("+*-/",e[i]) ) {
         s[j++] = ' ';
         while( !vaziap(P) && prio(topo(P))>=prio(e[i]) )
            s[j++] = desempilha(P);
         empilha(e[i],P);
      }
      else if( e[i] == ')' ) {
         while( topo(P)!='(' ) s[j++] = desempilha(P);
         desempilha(P);
      }
   while( !vaziap(P) ) s[j++] = desempilha(P);
   s[j] = '\0';
   destroip(&P);
   return s;
}
```

Figura 3.9 | Função para conversão de infixa em posfixa (com operandos reais de vários dígitos).

3.4.2 Função de avaliação

Seja `e` uma cadeia. Então, o endereço `e+i` representa o *sufixo* (ou subcadeia) de `e` que começa na posição `i` de `e`. Por exemplo, se `e` é a cadeia `"2 34 *1.79 5/+"`, `e+6` é a subcadeia `"1.79 5/+"`. Quando um sufixo de uma expressão posfixa inicia com um operando, podemos usar a função `atof()`, declarada em `stdlib.h`, para obter seu

valor real. Por exemplo, chamando `atof(e+6)`, obtemos o valor `1.79`. Depois disso, basta incrementar `i` para que ele avance até o fim desse operando.

A versão final da função de avaliação é apresentada na Figura 3.10.

```
float valor(char *e) {
   Pilha P = pilha(256);
   for(int i=0; e[i]; i++)
      if( isdigit(e[i]) ) {
         empilha(atof(e+i),P);
         while( isdigit(e[i+1]) || e[i+1]=='.' ) i++;
      }
      else if( strchr("+*-/",e[i]) ) {
         float y = desempilha(P);
         float x = desempilha(P);
         switch( e[i] ) {
            case '+': empilha(x+y,P); break;
            case '-': empilha(x-y,P); break;
            case '*': empilha(x*y,P); break;
            case '/': empilha(x/y,P); break;
         }
      }
   float z = desempilha(P);
   destroip(&P);
   return z;
}
```

Figura 3.10 | Função para avaliação da forma posfixa (com operandos reais de vários dígitos).

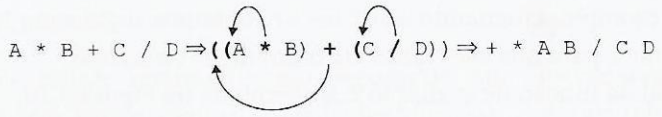
Exercícios

3.1 Use as funções das Figuras 3.5, 3.6 e 3.8 para criar um programa que exiba a forma posfixa e o valor de uma expressão infixa com inteiros de um dígito, digitada pelo usuário. *Dica*: uma pilha de inteiros pode guardar caracteres.

3.2 Use as funções das Figuras 3.5, 3.9 e 3.10 para criar um programa que exiba a forma posfixa e o valor de uma expressão infixa com reais de vários dígitos, digitada pelo usuário. *Dica*: uma pilha de reais pode guardar inteiros.

3.3 Supondo que os operandos em uma expressão infixa são representados por letras, crie uma função para converter essa expressão para a forma posfixa. Por exemplo, a conversão de `"A*B+C/D"` deve resultar em `"AB*CD/+"`. *Dica*: a função `isalpha(c)`, declarada em `ctype.h`, verifica se um caractere `c` é letra.

3.4 Na forma *prefixa* (ou *notação polonesa*), os operadores são colocados *antes* de seus operandos. Por exemplo, convertendo a expressão infixa `"A*B+C/D"` em prefixa, obtemos `"+*AB/CD"`. Para efetuar essa conversão, basta parentesiar completamente a expressão infixa e, depois, reescrevê-la, descartando os parênteses e movendo os operadores para a posição ocupada por seus parênteses esquerdos, como a seguir:



Supondo que os operandos em uma expressão infixa completamente parentesiada são letras, crie uma função para convertê-la em prefixa. *Dica*: a função `strlen(s)`, declarada em `string.h`, devolve o tamanho de uma cadeia `s` e a função `_strrev(s)`, declarada em `string.h`, inverte uma cadeia `s`.

3.5 Em C, *cadeia* de caracteres não é um tipo de dados *primitivo*, o que dificulta a sua manipulação. Para facilitar, podemos usar a função a seguir.

```
char *str(char *formato, ...) {
   char *s;
   vasprintf(&s, formato, (char *)&formato + sizeof(formato));
   return s;
}
```

Essa função é similar a `printf()`. Mas, em vez de exibir a saída em vídeo, ela a coloca num vetor de caracteres *dinâmico* e devolve seu endereço. Por exemplo, chamando `str("(%s%c%s)","um",'+',"dez")` obtemos `"(um+dez)"`. Considerando expressões cujos operandos são letras, use essa função para:

(a) Converter uma expressão *posfixa* em infixa completamente parentesiada.
(b) Converter uma expressão *prefixa* em infixa completamente parentesiada.

3.6 Considerando *expressões lógicas* infixas completamente parentesiadas, representadas por cadeias compostas exclusivamente por:

- *Operandos*, representados pelas letras `V` (*verdade*) e `F` (*falso*).
- *Operadores*, representados pelos caracteres `~` (*não*), `&` (*e*) e `|` (*ou*).
- *Parênteses*, que muda as prioridades dos operadores (`~`↦`3`, `&`↦`2` e `|`↦`1`).

(a) Crie uma função que devolve a *posfixa* de uma expressão lógica infixa.
(b) Crie uma função que devolve o *valor* de uma expressão lógica posfixa.
(c) Crie um programa que lê uma expressão lógica infixa e exibe sua forma posfixa e seu valor.