AVALIAÇÃO DE EXPRESSÕES

Os cientistas de computação ao conceberem a idéia de linguagens de programação de mais alto nível enfrentaram muitos obstáculos técnicos. Um dos maiores foi a questão de como gerar instruções em linguagem de máquina, que poderiam avaliar corretamente qualquer expressão aritmética. Uma declaração complexa de atribuição como

$$X \leftarrow A / B * C + D * E - A * C$$

poderia ter diversos significados e mesmo se fosse definida unicamente, digamos através do pleno uso de parênteses, pareceria ainda uma tarefa bastante complexa para gerar uma seqüência de instruções correta e viável. Felizmente a solução que temos hoje está tanto elegante como simples.

Uma expressão é formada de operandos e operadores. A expressão acima tem cinco operandos : A, B, C, D e E. Mesmo sendo estes todos variáveis de uma única letra, em uma linguagem de programação de alto nível os operandos podem ter qualquer nome de variável legal ou constante. Em qualquer expressão, os valores que podem assumir as variáveis devem ser consistentes com as operações desempenhadas com eles. Essas operações estão descritas pelos operadores. Na maioria das linguagens de programação há várias espécies de operadores que correspondem a diferentes tipos de dados assumidos por uma variável. Existem os operadores aritméticos básicos - adição, subtração, multiplicação e divisão. Outros operadores aritméticos incluem mais unário e menos unário. Em outra classe estão os operadores relacionais $<$, $\leq$, $=$, $\neq$, $\geq$ e $>$.

O primeiro problema com a compreensão do significado de uma expressão está em decidir em que ordem as operações devem ser executadas. Isto significa que cada linguagem deve definir singularmente tal ordem. Por exemplo, se A = 4, B = C = 2, D = E = 3, então na expressão anterior podemos querer atribuir a X o valor

```
4 / ( 2 * 2 ) + ( 3 * 3 ) - ( 4 * 2 )
( 4 / 4 ) + 9 - 8
2
```

Porém a verdadeira intenção do programador poderia ter sido atribuir o valor ao X

```
( 4 / 2 ) * ( 2 + 3 ) * ( 3 - 4 ) * 2
2 * 5 * -1 * 2
-20
```

Naturalmente ele poderia especificar esta última ordem de avaliação usando os parênteses.

$$X \leftarrow (A / B) * (C + D) * (E - A) * C$$

Para fixar uma ordem de avaliação, atribuímos uma prioridade a cada operador. Então entendemos que dentro de qualquer par de parênteses serão avaliados primeiro os operadores de mais alta prioridade. Especificadas as prioridades, a expressão $X \leftarrow A / B * C + D * E - A * C$ será avaliada como $X \leftarrow ((A / B) * C) + (D * E) - (A * C)$.
Como pode um compilador aceitar uma expressão assim e produzir um código correto ?

Um dos processos mais usados para a avaliação de expressões aritméticas consiste em primeiramente transformar a expressão de sua forma infixada (operadores entre operandos e uso de parênteses) para a notação *polonesa* pós-fixada (operadores após os operandos sem o uso de parênteses) e então avaliá-la com o auxílio de uma pilha.

O método baseia-se no fato de podermos avaliar, por exemplo, a expressão A + B + C * (D - E) através da seqüência de operações :

a) empilha A
b) empilha B
c) desempilha os dois últimos elementos e empilha a sua soma
d) empilha C
e) empilha D
f) empilha E
g) desempilha os dois últimos elementos e empilha a sua diferença
h) desempilha os dois últimos elementos e empilha o seu produto
i) desempilha os dois últimos elementos e empilha a sua soma

Toda vez que há referência explícita a um elemento o que temos a fazer é empilhá-lo e toda operação é feita sobre os dois últimos elementos empilhados. Assim, a seqüência de operações descrita acima está contida na forma A B + C D E - * +, que é o equivalente da expressão aritmética inicial na notação polonesa pós-fixada.

Veremos agora um algoritmo para transformação de uma expressão aritmética de sua forma infixada para a correspondente forma pós-fixada.

Esse algoritmo transforma uma expressão aritmética de sua forma infixada para a notação polonesa pós-fixada utilizando três listas. Uma fila **INFIX** que se supõe conter a expressão na notação infixada seguida de um símbolo especial λ; **POSFIX**, fila inicialmente vazia e que conterá a expressão final; e **TRAB**, pilha de trabalho utilizada para "lembrar" das operações que não puderam ser executadas até o momento. A pilha TRAB contém inicialmente outro símbolo especial π.

É assumida também a existência de uma função PRIOR que especifica as prioridades dos símbolos de acordo com a seguinte tabela

prioridade	**símbolos**
0	π
1	λ
2	(,)
3	+ , -
4	* , /
5	variáveis e constantes

```
ALGORITMO TransformaExpressãoInfixadaParaPós-Fixada
VARIÁVEIS
   FILA        : INFIX, POSFIX
   PILHA       : TRAB
   CARACTER    : X, Y
INÍCIO
   REPITA
      X ← INFIX
      SE PRIOR ( X ) = 5 ENTÃO { variável ou constante }
         X → POSFIX
      SENÃO
         SE X = ' ( ' ENTÃO
            X → TRAB
         SENÃO
            SE X = ' ) ' ENTÃO
               Y ← TRAB
               ENQUANTO Y ≠ ' ( ' FAÇA
                  Y → POSFIX
                  Y ← TRAB
               FIM ENQUANTO
            SENÃO { operador ou λ }
               ENQUANTO PRIOR ( X ) ≤ PRIOR ( "topo ( TRAB )" ) FAÇA
                  Y ← TRAB
                  Y → POSFIX
               FIM ENQUANTO
               X → TRAB
            FIM SE
         FIM SE
      FIM SE
   ATÉ X = λ
FIM
```

Os símbolos λ e π são usados no último enquanto; esse comando é utilizado para transferir operadores de prioridade maior que a do operador encontrado no momento da pilha de trabalho para a saída. O loop deve parar ao ser encontrado na pilha de trabalho um operador de menor prioridade que não pode ainda ser executado ou quando TRAB não contiver mais operadores. O símbolo π é usado para evitar o teste de *underflow* em TRAB. Uma situação especial ocorre quando terminou a expressão, ocasião em que todos os operadores em TRAB devem ser inseridos em POSFIX. O símbolo λ evita o loop adicional, além de tornar desnecessário o teste de *underflow* em INFIX.