Estrutura de Dados

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

4 de fevereiro de 2024

Aplicações em Pilhas

Análise da sintaxe de níveis de parêntesis

Considere a expressão:
$7 - ((X\dot{(}(X + Y)/(J - 3)) + Y)/(4 - 2,5))$

- Existe um número igual de parêntesis esquerdos e direitos.
- Todo parêntese da direita está precedido por um parêntese da esquerda correspondente.

Em uma análise se a expressão atende aos dois critérios podemos realizar uma análise com base em pilha.

Profundidade de Parêntesis

- Um parêntese de esquerda realiza uma abertura de nível.
- Um parêntese de direita realiza um fechamento de nível.
- A profundidade de nível em um determinado momento representa o número de parêntesis de esquerda abertos, mas ainda não fechados.
- O exemplo abaixo apresenta a contagem de nível para a expressão anterior

```
7 - ( ( X . ( ( X + Y ) / ( J - 3 ) ) + Y ) / ( 4 - 2,5 ) )
0 0 1 2 2 2 3 4 4  4  4 3 3 4 4 4 4 3 2 2  2 1 1 2 2 2  2  1 0
```

- A expressão pode ser mais complexa, com tipos diferentes de parêntesis: ([{...
- Neste caso, o fechamento de um nível deve ser correspondente ao mesmo símbolo que abriu, senão a expressão não é válida.

Análise de expressão com base em pilha

valid = verdade;
p = pilha vazia;
enquanto (não terminou expressão) {
  simb ← pega o próximo símbolo da expressão;
  se (simb = '(' **ou** simb = '[' **ou** simb = '{')
    empilhar(p,simb);
  se (simb = ')' **ou** simb = ']' **ou** simb = '}')
    se(ehvazia(p)) valid = falso;
    senão {
      s = topo(p); desempilhar(p);
      se(**não**(s complemento simb)) valid = falso;
    }
}
se(**não**(ehvazia(p))) valid = false;
se(valid) **imprime** "Expressão válida";
senão imprime "Expressão não válida";

Infixa, posfixa e prefixa

Modelo	Expressão
Infixa	$A + B$
Posfixa	$AB+$
Prefixa	$+AB$

Infixa, posfixa e prefixa

- A expressão infixa tem um formato visivelmente mais fácil de interpretar.
- A expressão infixa é ambígua, portanto é necessário estabelecer ordem de precedência, seja pela natureza da operação, seja pelo encapsulamento em parêntesis.
- As expressões posfixa e prefixa não são ambíguas, ao lado do operador estão dois operandos, se um deles é um operador, então este é realizado antes.
- As expressões posfixa e prefixa visualmente são mais complexas de se interpretar.

Ambiguidade

$$4/2/2$$

$$(4/2)/2 \neq 4/(2/2)$$

Precedência

- Considere as seguintes operações e ordem de precedência:
  1: $\wedge$ (exponenciação)
  2: * / (multiplicação e divisão)
  3: + - (soma e subtração)
- Índice menor da operação indica precedência sobre operações de índices maiores, exponenciação tem a maior precedência.
- Dentro do mesmo índice a precedência ocorre na ordem em que aparecem. Assim, 4/2/2 é interpretada como (4/2)/2.
- Exceto para a exponenciação em que a precedência é a ordem inversa: $A \wedge B \wedge C = A \wedge (B \wedge C)$
- Parêntesis sempre altera a ordem de precedência, as operações em parêntesis de nível mais interno devem ser realizadas primeiro.

Conversão: infixa $\rightarrow$ posfixa

- A conversão ocorre na ordem da operação. $AopB \rightarrow ABop$
- Exemplo:

$$A + (B * C)$$
$$A + (BC*)$$
$$A(BC*)+$$
$$ABC*+$$

- Não é necessário os parêntesis na forma posfixa

Conversão: infixa $\rightarrow$ posfixa

- A conversão ocorre na ordem da operação. $AopB \rightarrow ABop$
- Exemplo:

$$(A + B) * C$$
$$(AB+) * C$$
$$(AB+)C*$$
$$AB + C*$$

- Quando a precedência na infixa é diferente, na posfixa a forma é diferente.

Conversão: infixa $\rightarrow$ prefixa

$$A + (B * C)$$

$$+A * BC$$

$$(A + B) * C$$

$$* + ABC$$

Exemplos de Conversão

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

Exemplos de Conversão

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

Algoritmo de conversão usando pilha

- Lê-se uma expressão item a item
- Se for um operando ele é inserido direto na expressão posfixa
- Se for um abre parêntesis, ele é inserido na pilha
- Se for um fecha parêntesis, os operadores na pilha são removidos até o abre parêntesis
- Se for um operador:
  - Se no topo da pilha houver um operador de precedência menor, ou abre parêntesis, ele é inserido na pilha.
  - Senão (maior ou igual), o operador é removido até achar um de precedência menor, ou abre parêntesis ou pilha vazia. Então ele é inserido na pilha
- Se não houver mais itens na expressão, desempilha os operadores restantes.

Conversão - Exemplos

$$A + B * C$$

Lê	Posfixa	Pilha Operadores
A	A	
$+$	A	$+$
B	AB	$+$
*	AB	$+*$
C	ABC	$+*$
	$ABC*$	$+$
	$ABC*+$	

$$A * B + C$$

Lê	Posfixa	Pilha Operadores
A	A	
*	A	*
B	AB	*
$+$	$AB*$	$+$
C	$AB*C$	$+$
	$AB*C+$	

Conversão - Exemplos

$$A + B * C * D \wedge (E + F)$$

Lê	Posfixa	Pilha Operadores
A	A	
$+$	A	$+$
B	AB	$+$
*	AB	$+*$
C	ABC	$+*$
*	$ABC*$	$+*$
D	$ABC*D$	$+*$
$\wedge$	$ABC*D$	$+*\wedge$
(	$ABC*D$	$+*\wedge($
E	$ABC*DE$	$+*\wedge($
$+$	$ABC*DE$	$+*\wedge(+$
F	$ABC*DEF$	$+*\wedge(+$
)	$ABC*DEF+$	$+*\wedge$
	$ABC*DEF+\wedge$	$+*$
	$ABC*DEF+\wedge*$	$+$
	$ABC*DEF+\wedge*+$	

Calculadora posfixa

- É muito comum encontrarmos calculadoras baseadas em expressões posfixa para sua operação.
- A Hewlet Packard (HP) tradicionalmente tem uma linha de calculadora que opera no modelo posfixo.
- A evaluação de uma expressão posfixa ocorre em um modelo de pilha.
- Se um item é um operando, ele é inserido na pilha.
- Se um item é um operador, retira dois operandos do topo da pilha, realiza a operação e empilha o resultado.

Exemplo

$$623 + -382/ + *2 \wedge 3+$$

Lê	op1	op2	res	Pilha
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3				1,3
8				1,3,8
2				1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2				7,2
$\wedge$	7	2	49	49
3				49,3
+	49	3	52	52

Algoritmo de Calculadora

Algoritmo CALCULADORA(expr)
    Pilha S;
    enquanto expr $\neq$ vazia **faça**
        val $\leftarrow$ Proximo(expr)
        se Éoperador(val) **então**
            op2 $\leftarrow$ Desempilha(S)
            op1 $\leftarrow$ Desempilha(S)
            res $\leftarrow$ Calcula(op1,val,op2)
            Empilha(S,res)
        senão
            Empilha(S,val)
    val = Topo(S); Desempilha(S);
    retorne val

Além da calculadora: JVM

- A ideia baseada no uso de pilha para realizar operações, vai muito além de uma calculadora.
- Já existe modelo de processador baseado em operações em pilha.
- O JVM (Java Virtual Machine), máquina virtual java é um "processador" que tem toda a sua linguagem baseada em operações de pilha.
- As operações binárias: aritméticas, lógicas, condicionais, ... são realizadas sobre valores na pilha, e o resultado é empilhado.
- Outras operações transferem dados da pilha para memória e vice-versa.
- Mas este já é um tema de Arquitetura de Computadores e Sistemas Operacionais, logo, vale a pena entender bem pilhas.

O Merge Sort

- Uma sequencia é dividida em duas.
- Recursivamente, cada metade estará ordenada.
- O passo do algoritmo é intercalar ambas metades em uma única sequência ordenada.
- A recursividade termina quando o tamanho da sequência é um, $p >= r$.

O algoritmo recursivo: Merge_Sort

Entrada: Uma sequência de números A e os índices de início da sequência a ser ordenada: p e r

Saída: A mesma sequência, no intervalo $[p, r]$, de números, ordenada

Algoritmo $\mathrm{MERGE\_SORT}$(A,p,r)

se $p < r$ **então**

$q \leftarrow (p + r)/2$

MERGE_SORT(A, p, q)

MERGE_SORT$(A, q + 1, r)$

INTERCALA(A, p, q, r)

A função Intercala

- A primeira metade na sequência $[p, q]$ é copiada na primeira metade de um vetor de mesmo tamanho.
- A segunda metade $[q + 1, r]$ é copiada na segunda metade do vetor, mas em sequência inversa.
- Em seguida copia-se de volta intercalando os valores.
- Inicia-se das pontas, escolhendo entre as duas pontas, o menor valor.
- Constrói-se, assim, uma sequência única ordenada.

A função Intercala

Entrada: Uma sequência onde no intervalo [p,r] a sequência está ordenada no subarranjo [p,q] e no subarranjo [q+1,r].
Saída: A mesma sequência, ordenada no intervalo [p,r].

Algoritmo INTERCALA(A,p,q,r)

para $i \leftarrow p$ **até** q **faça**
    $B[i] \leftarrow A[i]$
para $j \leftarrow q+1$ **até** r **faça**
    $B[r+q+1-j] \leftarrow A[j]$
$i \leftarrow p$
$j \leftarrow r$
para $k \leftarrow p$ **até** r **faça**
    se $B[i] \leqslant B[j]$ **então**
        $A[k] \leftarrow B[i]$
        $i \leftarrow i+1$
    senão
        $A[k] \leftarrow B[j]$
        $j \leftarrow j-1$

A versão não recursiva, baseada em pilha

- Vamos trabalhar com duas pilhas, uma pilha para controlar as chamadas recursivas de divisão dos intervalos, e paralelamente, no mesmo nível recursivo, as chamadas de intercala.
- Para um intervalo (válido != base), dividimos a sequência em duas são duas chamadas recursivas.
- As duas chamadas recursivas implicam em empilhar duas sequências.
- Este intervalo também será tratado na intercala, empilhamos também para chamada futura.
- Quando não há mais divisões, todos os intervalos em que intercala será chamada estão na pilha do intercala.
- Basta desempilhar e chamar o intercala.

A versão não recursiva, baseada em pilha

```
void mergeSort(int *A, int p, int r) {
  pilha div, inter;
  int q;
  div.empilhar(interval(p,r));  // o intervalo que vamos dividir.
  while(!div.ehvazia()) {
    interval i = div.topo; div.desempilhar();
    p = i.getP(); r=i.getR(); q = (p+r)/2;
    if(r > p) { // Se ! p > r, não há o que fazer, base da recursão.
      inter.empilhar(i); // O intervalo que será dividido aguarda futura intercala.
      div.empilhar(interval(p,q));  // Novos intervalos a dividir.
      div.empilhar(interval(q+1,r));
    }
  }
  while(!inter.ehvazia()) { // A sequência de intervalos para a função intercala
    interval i = inter.topo(); inter.desempilhar(); // Estão empilhados. Resta
    p = i.getP(); r = i.getR(); q = (p+r)/2; // desempilhar e chamar a intercala.
    intercala(A,p,q,r);
  }
  return;
}
```

A versão não recursiva, baseada em pilhas: Comentários finais

- A recursividade nada mais faz do que definir uma ordem na qual os intervalos serão chamados na função intercala.
- A recursividade é substituida quando empilhamos esta ordem.
- O elemento que empilhamos: `interval` é um tipo abstrato que contém como informação os índices de início e fim do intervalo.
- Foi usado C++, pois podemos sobrepor o operador = para o tipo `interval`, assim na atribuição fazemos diretamente, não precisamos atribuir cada membro interno do tipo.
- Como o C++ também fica mais simples retornar um tipo abstrato na operação `topo`, pelo mesmo motivo.