Estruturas de Dados e Básicas I - IMD0029

Selan R. dos Santos

DIMAp – Departamento de Informática e Matemática Aplicada
Sala 25, ramal 239, selan@dimap.ufrn.br
UFRN

2016.1

Aplicações de Listas Lineares Sequenciais — Conteúdo

Aplicação de Pilhas: Expressões Aritméticas

- As **Filas** possuem o comportamento de armazenamento FIFO (*First In, First Out*) e, por isto, podem ser aplicadas para ajudar na solução de problemas de escalonamento ou ordenação.
- As **Pilhas** possuem o comportamento de armazenamento LIFO (*Last In, First Out*) e, por isto, podem ser aplicadas em vários algoritmos clássicos em compiladores.
- **Problema**: Como avaliar uma expressão atritmética, como "$3 + 2 * 5$"?
- *<demonstrar spotlight ou google search>*
- Expressões normalmente são codificadas em uma notação tradicional, que é considerada ambígua:
  - $3 + 2 * 5 = 25$ ou $= 13$?
  - solução é a utilização de regras de prioridade de operadores.
  - consequentemente a tarefa computacional torna-se mais complicada.

Aplicação de Pilhas: Expressões Aritméticas
Tipos de Notações

- Porém outros tipos de representação existem, considerando-se operações binárias:

Notação totalmente parentizada

Acrescenta-se sempre um par de parênteses a cada par de operandos e a seu operador. Também é conhecida como notação **infixa**.

Exemplo:
notação tradicional: $A * B - C/D$
notação parentizada: $((A * B) - (C/D))$

Aplicação de Pilhas: Expressões Aritméticas
Tipos de Notações

- Porém outros tipos de representação existem, considerando-se operações binárias:

Notação polonesa

Operadores aparecem imediatamente antes dos operandos. Esta notação explicita quais operadores, e em que ordem, devem ser calculados. Por esse motivo, dispensa o uso de parênteses, **sem ambiguidades**. Também conhecida como notação **prefixa**.

Exemplo:
notação tradicional: $A * B - C/D$
notação polonesa: $- * AB/CD$

Aplicação de Pilhas: Expressões Aritméticas
Tipos de Notações

- Porém outros tipos de representação existem, considerando-se operações binárias:

Notação polonesa reversa

É a notação polonesa com os operadores aparecendo após os operandos. Esta notação também explicita quais operadores, e em que ordem, devem ser calculados. Por esse motivo, dispensa o uso de parênteses, **sem ambiguidades**. Esta notação é tradicionalmente utilizada em máquinas de calcular. Também conhecida como notação **posfixa**.

Exemplo:
notação tradicional: $A * B - C/D$
notação polonesa: $AB * CD/-$

Aplicação de Pilhas: Expressões Aritméticas
Avaliação de Expressões

- Uma das tarefas básicas na área de compiladores, por exemplo, consiste em avaliar expressões aritméticas, normalmente representadas na forma infixa.
- Uma alternativa para facilitar a avaliação de expressões consiste em convertê-las da forma infixa para posfixa.
- Esta conversão tem as seguintes vantagens:
  - Durante o procedimento de conversão é possível detectar alguns erros de formação de expressão;
  - A expressão no formato posfixo **não** necessita da presença de delimitadores (parênteses) por ser uma representação não-ambígua, e;
  - O algoritmo para avaliar uma expressão posfixa é mais simples do que um algoritmo para avaliar uma expressão infixa diretamente.

Aplicação de Pilhas: Expressões Aritméticas
Avaliando Expressão Posfixa

- A avaliação automatizada é realizada com o auxílio de uma TAD pilha.
- Suponha que uma expressão $A * B + C - D$ na forma posfixa foi previamente armazenada em uma lista linear.
- Vamos analisar a expressão percorrendo-a sequencialmente da esquerda para a direita.

A	B	*	C	+	D	−

2	3	*	4	+	8	−

Aplicação de Pilhas: Expressões Aritméticas
Vantagens da Notação Posfixa

- Para facilitar a explicação do algoritmo de conversão entre notações vamos assumir que a expressão foi pré-processada, de tal forma que cada um de seus componentes (i.e. operando ou operador) encontra-se armazenado em um nó de uma lista linear.
- Neste pré-processamento também devem ser feitas avaliações sintáticas, de forma a eliminar erros básicos.
- Algumas observações facilitam a conversão da notação infixa para a posfixa:
  - Operandos aparecem na mesma ordem tanto na infixa quanto na posfixa.
  - Na posfixa os operadores aparecem na ordem em que devem ser calculados (esquerda → direita).
  - Operadores aparecem imediatamente após seus operandos.

Aplicação de Pilhas: Expressões Aritméticas
Definição da TAD Symbol

Uma possível interface para a TAD Symbol seria:

Definindo a TAD Symbol

```
   # Representa um símbolo, que pode ser um operador ou operando
1: tad Symbol
2:     var content: texto                  # armazena o conteúdo do símbolo em uma cadeia de caracteres
3:     var isOpnd: booleano                # se verdadeiro, indica que é operando; caso contrário é operador
4:     construtor ()                       # construtor que cria um símbolo vazio
5:         content ← ""
6:         isOpnd ← verdadeiro
7:     construtor ≡ função (texto, booleano)   # construtor que já inicializa um símbolo
8:     isOperand ≡ função () ⟶ booleano        # retorna V se operando, F caso contrário
9:     getValue ≡ função () ⟶ inteiro          # retorna inteiro correspondente ao operando
10:    getOperator ≡ função () ⟶ caractere     # retorna caractere correspondente ao operador
```


Aplicação de Pilhas: Expressões Aritméticas
Precedência de Operadores

- A principal preocupação do algoritmo de conversão é a ordem e a posição dos operadores.
- Como os parênteses da expressão infixa deverão ser eliminados algum tipo de "mecanismo" deve existir para armazenar os operadores temporariamente de forma a conservar sua precedência. **Qual estrutura usar?**
- Abaixo temos um exemplo de precedência de operadores

Precedência	*Operador*	*Associação*
1	()	⟶
2	^	⟵
3	* / %	⟶
4	+ -	⟶

Tabela: Precedência e ordem de associação de operadores em expressões aritméticas. %: resto da divisão inteira e ^: operação de exponenciação.

Aplicação de Pilhas: Expressões Aritméticas
Conversão Entre Notações (manualmente)

- Vamos tentar identificar, via exemplos, como o procedimento é realizado

$$A * B + C - D \qquad \underline{A} * \underline{B} + C -$$

Aplicação de Pilhas: Expressões Aritméticas

Conversão Entre Notações

- Vamos tentar identificar, via exemplos, como o procedimento é realizado

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

Tabela: Exemplos de equivalências entre forma infixa e posfixa.

Aplicação de Pilhas: Expressões Aritméticas

Avaliando Expressão Posfixa

- Relembrando... Precisamos analisar a expressão percorrendo-a sequencialmente da esquerda para a direita.

A	B	*	C	+	D	–

2	3	*	4	+	8	–

Aplicação de Pilhas: Expressões Aritméticas

Algoritmo: avaliar expressão no formato posfixo.

Entrada: Fila de 'Symbol' representando uma expressão no formato posfixo.
Saída: Resultado da expressão avaliada.

```
1:  função AvalPosfixa(FPosfixa: SymbolQueue): inteiro
2:      var symb: Symbol                          # símbolo atual a ser analisado
3:      var OPn: IntStack                         # pilha de operandos
4:      var opnd1, opnd2: Operando                # operandos auxiliares
5:      var resultado: inteiro                    # recebe o resultado de operação
6:      enquanto não FPosfixa.isEmpty() faça      # não chegar ao fim da fila...
7:          FPosfixa.dequeue(symb)
8:          se symb.isOpnd() então                # é operando?
9:              OPn.push(symb.getValue)           # empilha operandos
10:         senão
11:             OPn.pop(opnd2)                    # (inverter) recupera 2° operando
12:             OPn.pop(opnd1)                    # (inverter) recupera 1° operando
13:             resultado ← Aplicar symb à opnd1 e opnd2   # um "caso" para cada operador
14:             OPn.push(resultado)               # armazenar resultado; fila pode estar em processamento
15:     OPn.pop(resultado)                        # recuperar o valor final da pilha e...
16:     retorna resultado                         # ...retorna o valor inteiro do símbolo
```


Aplicação de Pilhas: Expressões Aritméticas

Algoritmo para Avaliação de Expressões, versão alto nível

- Até o momento o algoritmo para avaliar uma expressão aritmética pode ser sumarizado da seguinte forma:
  1. Expressão parentizada (em uma *string*?) tem seus elementos separados e armazenados em uma lista de componentes ou `Symbol` (operadores ou operandos)—análise sintática básica é realizada.
  2. Lista com expressão na forma infixa é convertida para a forma posfixa.
  3. Lista com expressão na forma posfixa é avaliada e resultado retornado.
- Portanto, falta ainda criar um algoritmo para o passo 2. **Sugestões?**

Aplicação de Pilhas: Expressões Aritméticas

Convertendo de Infixa para Posfixa

- Queremos um algoritmo que converta expressões:
  a. $A + B * C \Rightarrow ABC * +$
  b. $A * B + C \Rightarrow AB * C+$
- Com relação aos operandos: note que sua ordem é preservada.
- Com relação aos operadores:
  - Em algumas situações (a) a ordem dos operadores é invertida — primeiro a entrar é o último a sair, comportamento **FILO**: Pilha!
  - Em outras situações (b) a ordem dos operadores é preservada.
  - Quem determina estes comportamentos? Qual é o padrão?
  - Tente pensar, primeiramente, sem considerar os parênteses (que 'quebram' a ordem de precedência original).
- Precisamos de uma mecanismo (função) que, dados dois operadores, indique qual dos dois tem maior precedência
  - `prcd(op1,op2)`, onde `op1` e `op2` são operadores, retorna ***Verdadeiro*** se `op1` tiver a mesma ou maior precedência do que op2; ou ***Falso***, caso contrário—é equivalente a testar `op1` $\geq$ `op2`.

Aplicação de Pilhas: Expressões Aritméticas

Convertendo de Infixa para Posfixa

- Vamos simular um possível algoritmo com as expressões abaixo e tentar achar um padrão de comportamento que possa ser codificado:
  a. $A + B * C \Rightarrow ABC * +$
  b. $A * B + C \Rightarrow AB * C+$
  c. $A - B/C * D\hat{}E \Rightarrow ABC/DE\hat{} * -$
- Procure raciocinar utilzando as três informações abaixo:
  - ⋆ Há uma pilha de operações que pode receber (`push`) operadores;
  - ⋆ Existe uma função precedência de operações, e;
  - ⋆ É possível consultar o operador no topo da pilha.
- Lembre-se que o algoritmo recebe uma fila de `Symbol` e retorna uma nova fila de `Symbol` com os termos da expressão (infixa) original permutados de maneira a representar a mesma expressão na forma posfixa (saída).

Aplicação de Pilhas: Expressões Aritméticas

Algoritmo: converter expressão da forma infixa para posfixa (alto nível)

```
    Entrada: Fila de 'Symbol' representando uma expressão no formato infixo.
    Saída: Fila de 'Symbol' representando uma expressão no formato posfixa equivalente.
1:  função Infx2Posfx(fila no formato infixo): fila no formato posfixo
2:      enquanto não chegar ao fim da fila de entrada faça
3:          remover símbolo da fila de entrada em symb
4:          se symb for operando então
5:              enviar symb diretamente para fila de saída
6:          senão
7:              enquanto Pilha não estiver vazia e símbolo do topo (topSymb) ≥ symb faça
8:                  se topSym ≥ symb então
9:                      remover topSym e enviar para fila de saída
10:             Empilhar symb     # depois que retirar operadores de precedência ≥, inserir symb
        # descarregar operadores remanescentes da pilha
11:     enquanto Pilha não estiver vazia faça
12:         remover símbolo da pilha e enviar para fila de saída
13:     retorna fila de saída na forma posfixa
```


Aplicação de Pilhas: Expressões Aritméticas

Algoritmo: converter expressão da forma infixa para posfixa

```
    Entrada: Fila de 'Symbol' representando uma expressão no formato infixo.
    Saída: Fila de 'Symbol' representando uma expressão no formato posfixa equivalente.
1:  função Infx2Posfx(FInfixa: SymbolQueue): SymbolQueue
2:      var symb, topSym: Symbol                                  # símbolos atual a auxiliar
3:      var OPr: SymbolStack                                      # pilha de operadores
4:      var FPosfixa: SymbolQueue                                 # receberá a expressão convertida
5:      var precdPilhaMaiorEntrada: booleano                      # testa precedência entre operadores
6:      enquanto não FInfixa.isEmpty() faça                       # não chegar ao fim da fila de entrada...
7:          FInfixa.dequeue(symb)
8:          se symb.isOpnd() então                                # é operando?
9:              FPosfixa.enqueue(symb)                            # operandos vão direto para saída
10:         senão
11:             precdPilhaMaiorEntrada ← verdadeiro               # possibilitar a entrada no laço
12:             enquanto não OPr.isEmpty() e precdPilhaMaiorEntrada faça   # pilha não-vazia
13:                 OPr.top(topSymb)
14:                 se (precdPilhaMaiorEntrada ← prcd(topSymb, symb)) então # topo ≥ simb
15:                     OPr.pop(topSymb)                          # sai da pilha e...
16:                     FPosfixa.enqueue(topSymb)                 # ...vai para saída
17:             OPr.push(symb)    # depois que retirar operadores de precedência ≥, inserir simb
18:     enquanto não OPr.isEmpty() faça # descarregar operadores remanescentes da pilha para a posfixa
19:         OPr.pop(topSymb)                                      # removendo operadores restantes
20:         FPosfixa.enqueue(topSymb)
21:     retorna FPosfixa                                          # retorna fila resultante
```

- Para pensar e resolver:
  - **Como quebrar a expressão em componentes e fazer a análise sintática?**
    - Definir uma gramática EBNF (Extended Backus-Naur Form) e usar um *recursive descent parser*

```
1 expression = term { ("+"|"-"|"%"|"*"|"/"|"^") term }.
2 term = [ "+"|"-" ] number | "(" expression ")".
3 number = {("0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9")}.
```

    - Definir uma gramática livre de contexto e usar um *LL parser* (baseado em pilhas).
  - **Como seria tratado os parênteses?**
  - **Como modificar o algoritmo para dar suporte ao '-' unário?**
    - Ex.: $-A + C * D$ ou $A * (-B)$.
  - **Implementar o resto da TAD Symbol**

Referências

J. Szwarcfiter and L. Markenzon.
Estruturas de Dados e Seus Algoritmos, 2ª edição, **Cap. 2**.
Editora LTC, 1994.

A. M. Tenembaum, Y. Langsam e M. J. Augenstein.
Estruturas de Dados Usando C, 1ª edição, **Cap. 3**.
Editora Pearson, 1995.