Relatório - Trabalho 1
Avaliação de Expressões Aritméticas (forma posfixa)

Thiago Veras Machado, 16/0146682
Vitor Fernandes Dullens, 16/0148260

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CiC 116319 - Estrutura de Dados - Turma A

{thiagomachado,vitordullens}@cjr.org.br

1. Introdução

1.1. Descrição

O trabalho consiste em uma calculadora de expressões, no qual a entrada vai ser uma expressão no tipo infixa (i.e: (2+4)*3), e o programa deverá ler a expressão digitada pelo usuário, analisar se é uma expressão válida, tranformá-la para a forma posfixa, utilizando a estrutura de dados pilha, além de mostrar na tela para o usuário ela(expressão) na forma posfixa e mostrar o resultado da expressão.

1.2. Visão Geral

O programa vai ler do teclado, atravéz da função *scanf* o que for digitado pelo usuário, necessariamente uma expressão infixa apenas com números inteiros, irá verificar a validade da expressão utilizando casos testes, utilizará uma pilha para transformá-la da forma infixa para a posfixa e, após isso, calculará o resultado da expressão conforme as prioridades exigidas pelos parênteses (i.e: (2+3)*4 ; será calculado primeiro a soma, devido aos parênteses que possuem prioridade), depois disso, imprimirá na tela a forma posfixa e o resultado através da função *printf*.

2. Implementação

2.1. Estrutura de Dados

A estrutura de dados utilizada no programa foi a pilha, mais especificamente a pilha estática, na qual existem, assim como na pilha dinâmica, duas funções principais; push e pop.

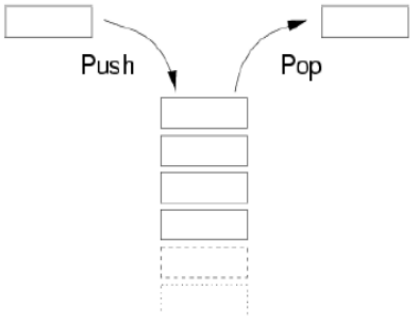

Figura 1. Funções de uma pilha

Essa funções são as responsáveis por empilhar (push) e desempilhar (pop) elementos da pilha. Como é ilustrada na Figura 1 a pilha é uma estrutura de dados que se assemelha a uma fila, só que o último elemento a entrar é o primeiro elemento a sair (LIFO = Last In, Firt Out).
Em nosso programa essas funções foram implementadas desta maneira:

```
void push(char num){
    pilha[++topo] = num;
}
char pop(){
    if(topo == -1) return printf("ESTA VAZIA\n");
    return pilha[topo--];
}
```

A função *push(char)* é responsável por colocar o caracter passado como parâmetro no topo da lista. A variável global topo é o índice da pilha e é responsável por mostrar onde será retirado o elemento e onde será inserido o próximo. Já a função *pop()* é responsavel por remover o elemento do topo da pilha e retorná-lo. Como é utilizado pilha estática(vetor), a função só volta com o índice (topo–), fazendo com que o próximo elemento, quando for armazenado pela *push(char)*, sobrescreva o valor que estava inserido. Porém se a função for chamada e a lista estiver vazia, é retornado para o usuário que a pilha está vazia.

```
char pilha[1000];
int topo = -1;
double pilha_r[1000];
int topo_r = -1;
```

Acima estão as representações das pilhas estáticas em nosso programa, implementadas como variáveis globais, sendo seu tamanho fixo e o *topo* sempre começando por -1, que representa a pilha vazia. A *char pilha* é a pilha de caracteres, incluindo letras e operadores, e a pilha *double pilha_r* é a que armazena os números para serem realizadas as operações. Essa pilha que armazena os números possui suas próprias funções de push (*push_(double)*) e pop (*pop_r*), além de possuir o índice próprio (topo_r).

2.2. Principais Funções

```
char check(){
    return pilha[topo];
}
```

check() e *check_r()*: é uma função auxiliar de pilha extremamente importante para o desenvolvimento do programa, ele é responsável apenas por retornar o valor que está no topo da pilha, o que é essencial para algumas outras funções.

```
void to_numbers(char infixo[]){
        int i, k = -1;
        char aux[1000];
        for( i = 0 ; infixo[i] != '\0'; ++i){
                if(isdigit(infixo[i])) aux[++k] = infixo[i];
                else if(k>= 0){
                        aux[++k] = '\0';
                        numeros[++pos] = atoi(aux);
                        k = -1;
                }
        }
}
```

to_numbers(): Esta função foi fundamental para a realização da função *resolve()*. Ela utiliza uma string como parâmetro, no formato infixo, realiza o processo de capturação de todos os números contidos na expressão e salva-os em um vetor global para facilitar o cálculo posteriormente. A função basicamente percorre a string através de um laço, verifica se o caracter é número (a partir da função *isdigit*), e caso seja verdade, é passado o caracter para uma string auxiliar. Caso seja falso, significa que acabou a quantidade de dígitos do número, então o programa acrescenta o contra barra 0 para fechar a string e a transforma para inteiro (utilizando a função *atoi*) e reseta o index da string auxiliar, fazendo com que só seja possivel entrar no *else if* (onde transforma para inteiro) se tiver algum dígito na string auxiliar.

```
bool validar()
```

Validar(): Foi realizada a partir de 4 inteiros: N, que representa o tamanho da expressão, i, para o laço e L e R para contabilizar a quantidade de parênteses abrindo e fechando. Basicamente a função percorre pela expressão entre os 2 parênteses pré colocados, verifica se o caracter em questão é '(', se for ele incrementa o L e inicializa um *while* até atravessar todos os '(', após atingir todos ele verifica se o próximo caracter é um número ou uma letra, pois são os únicos possíveis após um '(', caso tudo estiver Ok, o programa continua pulando todos os '(' e o próximo caracter, pois tudo já estava válido e nao é necessário computá-los novamente. Caso o caracter seja ')', o mesmo *while* é executado e o programa verifica se existe próximo caracter (caso tenha acabado a expressão), se tiver, ele verifica se é um operador. Caso a função não tenha ')' ou '(', o programa ainda necessita computar os operadores e faz uma verificação se o caracter anterior é alfanumérico ou ')' e se o caracter posterior é alfanumérico ou '('. O penúltimo teste está relacionado ao tipo do caracter, caso não seja nenhum dos outro e também não seja alfanumérico, a entrada está inválida. Caso passe em todos os testes, o programa irá retornar verdadeiro se o numero de parênteses abrindo é o mesmo dos parênteses fechando, que é o ultimo teste feito.

```
void in2pos(char infixo[])
```

in2pos(): Esta função tem o papel de transformar a expressão infixa para posfixa. Sua implementação é bem simples, feita com o auxílio de 2 inteiros indexadores (i e l), um caracter auxiliar (aux), uma pilha de caracteres para empilhar os operadores e parênteses e um vetor de caracter global de caracteres para salvar todos os operadores para facilitar a função resolve(). É iniciado um laço que percorre toda a entrada, utilizamos a função

switch para facilitar a leitura do código, caso seja um '(', apenas o empilhamos, caso seja ')' iremos desempilhar e imprimir na tela todos os operadores presentes na pilha até atingir o '(' (pois é onde termina a subexpressão) e depois removemos o '('. Durante o processo de impressão do operador, o vetor global "posfixo"recebe o operador. Caso o caracter seja '+' ou '-', desempilhamos tudo que está na pilha e imprimimos até o '(', depois empilhamos esse operador. O mesmo serve para o '/' e o '*', porém a diferença é que se imprime tudo que está na pilha enquanto não encontra o '(' e que seja diferente dos operadores anteriore ditos ('+' e '-'), o caso base do switch é apenas imprimir na tela o número da entrada ou a letra, caso seja numero, é executado um *while* imprimindo todos os dígitos do número.

```
1 void resolve()
```

resolve(): A função resolve tem o papel, como o próprio nome já diz, de resolver a expressão imposta. Ela é executada apenas se a expressão for válida. Sua implementação foi realizada a partir de 2 indexadores inteiros (i e k), uma string aux, uma variável boolena, uma pilha para os resultados (pilha_r) e o vetor global que contém apenas os números na ordem da expressão, pré computados na função *to_numbers()*. Inicia-se um laço que percorre todos os caracteres do vetor de caracter global "posfixo"já computados durante a função *in2pos*, caso o caracter em análise seja um número, nós empilhamos o primeiro valor do vetor global de números e incrementamos o indexador para pegar o próximo, caso necessário. Após exibir na tela o número, nos deparamos com um problema de que caso o número empilhado tenha mais de 1 dígito, pois o próximo caracter do laço poderia ser o segundo dígito do número, que já foi computado na pilha, para isso, nós passamos o número para string aux, utilizando a função *sprintf*, e utilizamos a função *strlen* para saber a quantidade de dígitos, e assim incrementamos o indexador do laço para continuar a análise após o último dígito do número. Se o caracter em análise for um operador, o programa utiliza a função *switch* para auxiliar nas operações. Desempilhamos 2 números da pilha, no qual serão computados a partir do operador, após realizado o cálculo o número resultante é empilhado novamente. O único diferencial está na divisão, na qual o programa analisa se o divisor é zero, caso seja verdade, a variável booleana ok é setada como falsa e as operações são abortadas. Para a impressão na tela do resultado, é analisado previamente a variável ok, caso seja verdade nos diz que foi possível computar e imprime o topo da pilha, pois após todas a osperacões restará apanas um elemento na pilha.

3. Estudo de Complexidade

A implementação foi feita com um foco em sua otimização, assim sendo, adotamos algumas funções que facilitariam e otmizariam o código. Sobre as funções da pilha estática, todas são de complexidade O(1), assim como a função *Ler( )*, que executa apenas 4 operações. Em relação às funções passadas, não se tem uma implementação difícil para entender sua complexidade, diferente das demais funções que serão analisadas nesse tópico:

1. *void to_numbers(char infixo[])*: Esta função possui complexidade linear, sendo n o tamanho da string passada como parâmetro, pois executa apenas um laço e

a quatidade de execuções é o número de caracteres, como dentro do laço possui apenas 2 condicionais que são 0(1), a função é linear.

2. *validar()*: É a função mais complexa de todo o programa, porém sua complexidade é linear. A função contém um laço principal, que executa n operações (assim como o anterior), dentro dele há mais 2 laços, o interessante dessa função é que mesmo com os *while* dentro do laço principal, a função ainda é linear. Sua linearidade é obitida com o fato de que após todas checagens dos '(' ou ')', como explicada anteriormente, a função pula todos os parênteses verificados, fazendo com que o laço não faça uma nova comparação de um parênteses já verificado, tornando-se assim, linear.
3. *in2pos()*: Possui complexidade linear, é feita a partir de um laço que executa n operações (igual as anteriores) e cada um dos *while* desempilha cada um dos operadores já empilhados enquanto não encontrar um '(', isso faz com que, no pior dos casos, o programa execute n + x operações (n = numero de caracteres da expressão infixa, e x = número de operadores), sendo assim, O(n+x) que assintóticamente é O(n), ou seja, linear.
4. *resolve()*: A função *resolve()* possui apenas um laço que executa n operações (igual as anteriores). O interessante dessa função é que após o programa reconhecer o primeiro dígito de cada número, ele pula para seu último dígito devido à função *void to_numbers()* que já possui todos os números em formato int, fazendo com que o i seja incrementado com a quantidade de dígitos do número. Sendo assim, a complexidade desta função O(x+y), sendo 'x' a quantidade de caracteres não numéricos e 'y' a quantidade de números, que assintóticamente é O(n), ou seja, linear.

4. Testes Executados

Testes para verificar se a forma posfixa e o resultado das expressões está correto:

- a*(b+c)*(d-g)*h : Posfixa = abc+*dg-*h* . O programa deve mostrar apenas a forma posfixa;
- (a-b)*c : Posfixa = ab-c* . O programa deve mostrar apenas a forma posfixa;
- a-b*c : Posfixa = abc*- . O programa deve mostrar apenas a forma posfixa;
- (1)+(1) : Resultado = 2; Posfixa = 1 1 + . O programa deve mostrar a forma posfixa e o resultado da expressão;
- 3-1*2+3-1 : Resultado = 3; Posfixa = 3 1 2 * - 3 + 1 - . O programa deve mostrar a forma posfixa e o resultado da expressão.

Testes para verificar a validez da expressão:

- (1+1)) : Função inválida, pois possui mais parênteses que fecham do que os que abrem ou vice-versa;
- () : Não possui nenhuma operação a ser feita, o valor é nulo, ou seja, não é uma expressão infixa, com isso, não pode-se calcular sua posfixa, muito menos seu resultado;
- x/0 : Qualquer divisão por zero da um valor indefinido/incalculavel, pois zero é um valor nulo, por isso, a expressão é invalida.
- (1++1) ou (1)+(+1) : São expressões que possuem dois operadores seguidos, o que torna a expressão inválida, já que não está na forma usual;
- (1)(1) : Expressão que não possui o operador definido também é dada como inválida, visto que não é está na forma usual de escrita de uma expressão.

5. Conclusão

5.1. Comentários Gerais

Podemos concluir que com esse trabalho foi abordado, de uma melhor forma, o conceito de pilhas e aspectos sobre a implementação das mesmas, mais precisamente da pilha estática, como visto na Seção 2. Também foi adquirido melhor conhecimento sobre a análise de algoritmos/estudo de complexidade, no qual, após todas as análises de cada função, concluímos que o programa é linear, visto que foi importante pensar em qual algoritmo implementar a fim de que nosso código ficasse otimizado e compreensível.

Portanto, foi uma ótima experiência, que nos trouxe um amadurecimento sobre o assunto de pilhas e listas, além de nos colocar para praticar a criação de programas bem elaborados e que devem conseguir exercer suas funções sem erros.

5.2. Principais Dificuldades

Em nossa visão, a parte mais complexa do trabalho foi a validação da expressão, mais precisamente a funcão *validar()*, pois são muitos casos testes a serem verificados e atribuidos ao código sem fazer com que outras expressões que são válidas sejam reconhecidas como inválidas devido a algum erro gerado após a implementação. Nos vimos, as vezes, tornando as expressões que eram válidas, inválidas, tentando implementar um caso em que a função seria inválida.