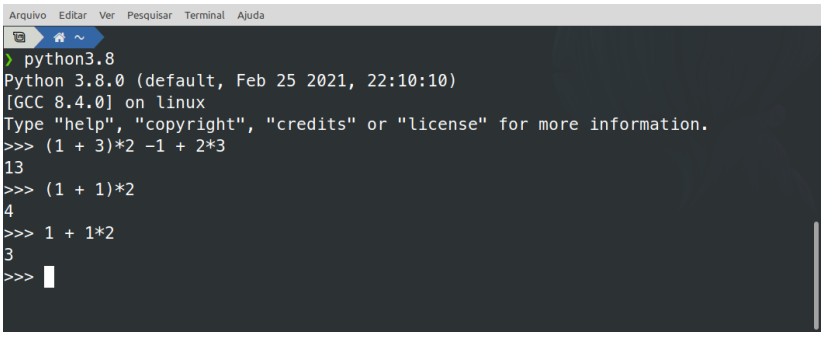
Universidade Federal do Espírito Santo

Departamento de Computação e Eletrônica - CEUNES
Estrutura de Dados I
Prof. Oberlan Romão

Exercício Programa 2

Sucuri[1]

Uma das características de Python é que com uma única linha de código podemos resolver expressões matemáticas e nem precisamos fazer um arquivo para isso, basta usarmos o Shell do Python (como mostrado na imagem abaixo).



Nesse EP, sua tarefa é implementar uma versão brasileira do Shell do Python, a Sucuri, que resolve expressões matemáticas. Cada expressão pode conter números, abre e fecha parênteses e os seguintes símbolos correspondentes a 7 operadores aritmético:

Operação	Símbolo
Exponenciação	^
Resto de divisão	%
Multiplicação	*
Divisão	/
Adição	+
Subtração	-
Menos unário	_

Baixe o arquivo `Sucuri.zip` e extraia o conteúdo em alguma pasta. Analise o conteúdo de cada arquivo. Alguns arquivos não devem ser modificados, enquanto outros estão parcialmente implementados e você deve completá-los.

[1]Baseado em um trabalho da professora Cristina G. Fernandes (IME-USP)

Tarefa

Após a leitura de uma expressão, a função `criaFilaObjetos` (`Lexer.c`) faz uma análise léxica dos elementos da linha, ou seja, ela fica responsável por "separar" cada item (operando ou um operador ou abre/fecha parênteses) em uma fila de objetos (veja o arquivo `Objetos.h`), onde cada objeto armazena um item lido. Um objeto é uma estrutura da seguinte forma:

```
typedef union valor {
    int    vInt;
    double vFloat;
    char   *pStr;
} Valor;

typedef struct objeto {
    Categoria categoria;
    Valor valor;
    struct objeto *proximo;
} Objeto;
```

Note que o campo `valor` é uma variável genérica que pode armazenar ou um `int` ou um `double` ou um `char *`. O campo `categoria` (definido no arquivo `Categoria.h`) indica o que o objeto está armazenando. Por exemplo, um operador de multiplicação ou um operador de adição, etc. Assim, inicialmente, um objeto é um par da forma: `(item, categoria)`, onde `item` é um string representando o item léxico encontrado.

Ao final da análise léxica, temos uma fila de pares da forma `(item, categoria)`. Por exemplo, se a linha lida contiver a expressão `2+3.5`, a análise léxica produzirá os pares:

```
("2", INT_STR)
("+", OPER_ADICAO)
("3.5", FLOAT_STR)
```

Em seguida, cada um desses itens (strings) deve ser substituído por um valor e assim obteremos uma fila de pares da forma `(valor, categoria)`. Dessa forma, na próxima fase esses strings são substituídos por números da seguinte forma:

- se a categoria do item é `INT_STR` (string representando um int) então o item do par é substituído pelo `int` correspondente;
- se a categoria do item é `FLOAT_STR` (string representando um float) então o item do par é substituído pelo `double` correspondente;
- se a categoria de item corresponde a um operador, o item do par é substituído por um número inteiro representando a precedência do operador (veja o arquivo `Util.c`).

Para o exemplo de pares anterior, o resultado dessa substituição é:

```
(2, INT)
(1, OPER_ADICAO)
(3.5, FLOAT)
```

A função de alta ordem `converteElementosFila` (`Fila.c`) é a responsável por fazer essa conversão. A função deve receber uma fila e uma função que faz a conversão de cada objeto. Você deve implementar a função `converteElementosFila` e chamá-la passando a fila de objetos e a função `itemParaValor`, que já implementada no arquivo `main.c`.

Notação infixa para pós-fixa

Após a conversão dos objetos para pares da forma `(valor, categoria)`, a função `infixaParaPosfixa` (do arquivo `Posfixa.c`) é chamada para converter a expressão em notação infixa para notação pós-fixa. Na notação tradicional (notação infixa) temos que o operador aparece entre os seus dois operandos. Já na notação pós-fixa, também conhecida como notação polonesa inversa, o operador é colocado após os seus dois operandos. Veja alguns exemplos:

infixa	**pós-fixa**
$10 - 20$	$10\ 20\ -$
$13 - 2 * 5$	$13\ 2\ 5\ *\ -$
$(1 + 3) * 2$	$1\ 3\ -\ 2\ *$
$1 + 2 * 3\ \hat{}\ 4 - 5$	$1\ 2\ 3\ 4\ \hat{}\ *\ +\ 5\ -$

Note que os operandos aparecem na mesma ordem na expressão infixa e na correspondente expressão pós-fixa. Note também que a notação pós-fixa dispensa parênteses e regras de precedência entre operadores (como a precedência de $*$ sobre $-$ por exemplo), que são indispensáveis na notação infixa. Na notação pós-fixa, a ordem dos operadores na expressão diz a ordem em que eles vão ser executados (da esquerda para a direita).

A função `infixaParaPosfixa` deve receber uma fila de objetos contendo a expressão infixa e retornar uma fila com a expressão correspondente na notação pós-fixa. O pseudo-código abaixo ilustra uma forma que podemos usar para converter uma expressão infixa para pós-fixa.

- Crie uma pilha vazia para manter os operadores.
- Crie uma fila vazia para a saída.
- Examine cada objeto da fila `infixa` e se o objeto for:
  - um operando (`FLOAT` ou `INT`), coloque-o na fila de saída.
  - um abre parêntese (`ABRE_PARENTESES`), insira-o na pilha.
  - um fecha parênteses (`FECHA_PARENTESES`), remova os objetos da pilha até que o abre parêntese correspondente seja removido. Coloque cada operador removido na fila de saída.
  - um operador insira-o na pilha. Entretanto, remova antes os operadores que estão na pilha que têm precedência maior ou igual ao operador encontrado e coloque-os na fila de saída. Lembre-se que o campo `valor` de um objeto que armazena um operador contém o valor da sua precedência (quanto maior esse valor, maior é a precedência do operador).
- Quando a expressão tiver sido completamente examinada, verifique a pilha. Qualquer operador que ainda está na pilha deve ser removido e colocado na fila de saída.

A Figura 1 ilustra os passos do algoritmo para converter a expressão infixa $A * B + C * D$. Note que o primeiro $*$ é removido assim que o operador $+$ é encontrado, já que a multiplicação tem precedência maior que a adição. Por outro lado, o operador $+$ permanece na pilha quando o segundo $*$ ocorre, uma vez que a adição tem menor precedência. Ao final da expressão infixa removemos da pilha ambos operadores colocando-os como últimos operadores da expressão pós-fixa.

Dica: as funções `dequeue` e `desempilha` devem desalocar os objetos (chamando a função `liberaObjeto`) que estavam na fila e pilha, respectivamente. Para evitar problemas de falha de segmentação, faça uma cópia de cada objeto (`copiaObjeto`) que será inserido na pilha e/ou fila de saída da função `infixaParaPosfixa`.

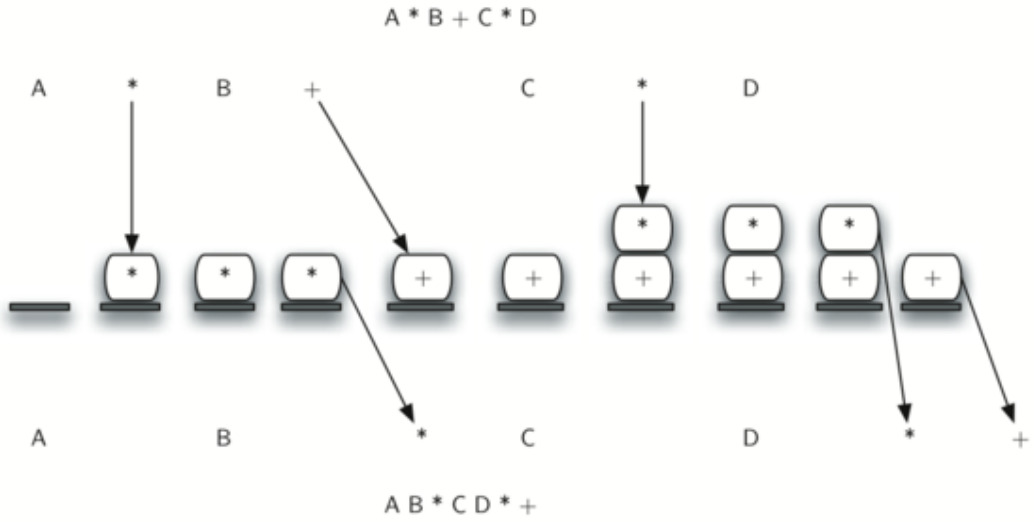


Figura 1: Convertendo $A * B + C * D$ para notação pós-fixa [Fonte: Link]

Se executarmos o `Sucuri` com a opção "`-e`", ele irá mostrar o resultado da conversão da expressão infixa para pós-fixa. Veja um exemplo (as expressões sublinhadas foram digitadas):

```
./Sucuri -e
Estrutura de Dados 1 (2020/2) - EP2
Sucuri 1.0.0 (Apr  6 2021, 18:41:11)
[GCC 7.5.0] on Linux
>>> 1+2
Expressão pós-fixa: 1 2 +
3
>>> (1+2)*3
Expressão pós-fixa: 1 2 + 3 *
9
>>> 3^2^2
Expressão pós-fixa: 3 2 ^ 2 ^
81
>>> 1+2*3^4-5
Expressão pós-fixa: 1 2 3 4 ^ * + 5 -
158
```

Precedência entre operadores

A tabela abaixo[2], apresenta a ordem decrescente de prioridade dos operadores: os operadores da primeira linha são executados em primeiro lugar e os operadores da última são executados por último. Assim, os operadores '+' e '−' possuem a menor precedência dentre os operadores da tabela. A coluna da direta indica a convenção de associação para os operadores da linha.

[2]Baseada na tabela de precedência disponível nesse link: https://www.ime.usp.br/~pf/algoritmos/apend/precedence.html

( )	esquerda-para-direita
_ ^	direita-para-esquerda
$*$ / %	esquerda-para-direita
$+$ $-$	esquerda-para-direita

Exemplos:

Expressão	**Interpretação**	**Resultado**
$2+3+4$	$(2+3)+4$	9
$2+3*4$	$2+(3*4)$	14
$2*3/4$	$(2*3)/4$	1
2^3^2	2^(3^2)	512
$2*$_3^4 -2	((2 * (_(3^4)) -2)	-164

Note que os operadores '_' (menos unário) e '^' (exponenciação) possuem a mesma precedência, mas são analisados da direita-para-esquerda. Diferente, por exemplo, dos operadores '$+$' e '$-$' que são analisados da esquerda-para-direita.

Interpretação da expressão pós-fixa

Após converter para a notação pós-fixa, a função `avalia` (`Avalia.c`) é chamada para avaliar e calcular o valor da expressão. A função recebe uma fila (`posFixa`) de objetos na notação pós-fixa. Para calcular o valor da expressão, a função `avalia` deve utilizar uma pilha, a chamada pilha de execução. A sua função deve examinar cada objeto da fila `posFixa` e:

- Se o objeto for um operando (`FLOAT` ou `INT`), `avalia` deve empilhá-lo;
- Se o objeto contém um operador, `avalia` deve:
  - desempilhar um ou dois números da pilha, dependendo do tipo do operador;
  - calcular o valor da operação correspondente; e
  - empilhar o valor calculado.

Ao final, a função deve retornar o objeto do topo da pilha (ou uma cópia dele). Novamente, faça cópia dos objetos para evitar vazamento de memória e/ou falha de segmentação.

Vamos a um exemplo, considere a expressão pós-fixa `4 5 6 * +`. Ao examinarmos a expressão da esquerda para a direita, encontramos primeiro os operando 4 e 5. Neste ponto, ainda não temos certeza do que fazer com eles até ver o próximo objeto (que pode ou não ser um operador). Colocando-os em um pilha garantimos que eles estejam disponíveis se um operador vier em seguida. No exemplo, o próximo objeto é outro operando. Então, novamente o inserimos na pilha e analisamos o próximos objeto. Agora temos o operador $*$. Isso significa que os dois operando mais recentes devem ser multiplicados. Removendo os dois itens da pilha, podemos obter os dois operando apropriados e realizar a multiplicação (obtendo o resultado 30). Colocamos esse resultado na pilha para que ele possa ser usado como um operando dos operadores posteriores na expressão. O próximo objeto é o operador `+`, novamente desempilhamos os dois

operandos da pilha, calculamos a soma deles e empilhamos o resultado. Se a expressão e o calculo estiver correto, quando o operador final é processado, haverá apenas um objeto restante na pilha. Este representa o resultado da expressão. A Figura 2 mostra o conteúdo da pilha ao processarmos a expressão `4 5 6 * +`.

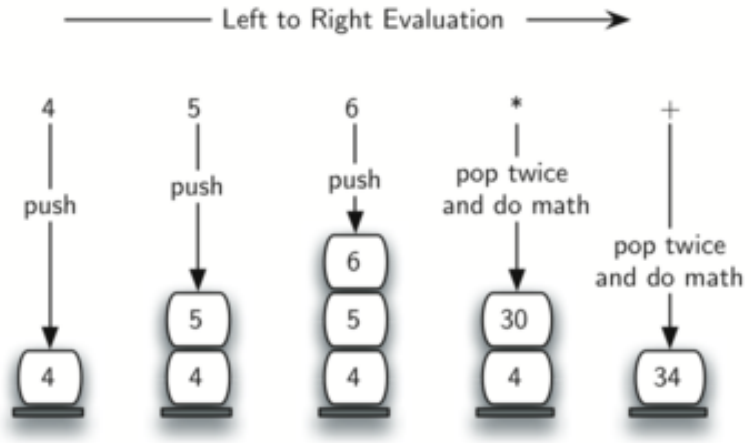


Figura 2: Conteúdo da Pilha durante a avaliação da expressão `4 5 6 * +` [Fonte: Link]

O `Sucuri` deve ter o mesmo comportamento da linguagem C ao resolver uma expressão. Assim, `FLOAT OPERADOR FLOAT` ou `FLOAT OPERADOR INT` deve resultar em um `FLOAT`. Por outro lado, `INT OPERADOR INT` deve resultar em um `INT`. Por exemplo, `1/2` resulta em `0` enquanto que `1.0/2` ou `1/2.0` resulta `0.5`.

Modo interativo ou script

O programa `Sucuri` pode ser usado no modo interativo (*default*) ou no modo script. Como em Python, no modo interativo, um prompt (**>>>**) é apresentado no início de cada linha para que a expressão seja digitada. Veja um exemplo (as expressões sublinhadas foram digitadas):

```
./Sucuri
Estrutura de Dados 1 (2020/2) - EP2
Sucuri 1.0.0 (Apr  6 2021, 17:15:04)
[GCC 7.5.0] on Linux
>>> 1+2
3
>>> (1+2)*3
9
>>> 3^3
27
>>> 3^2+1
10
>>> 3^(2+1)
27
>>> 3^2^2
81
```

Para executar o `Sucuri` no modo *script*, devemos passar como argumento a opção "-s" seguido do nome do arquivo que contém a(s) expressão(ões). Dessa forma, se quisermos resolver as expressões do arquivo `expressoes.txt`, basta executarmos o comando "`./Sucuri -s expressoes.txt`". Por exemplo, se o conteúdo do arquivo `expressoes.txt` for:

```
2^3+1
2^(3+1)
1/2
1.0/2
_1
3^2^2
2*_3^4-2
_3^2
(_3)^2
```

teremos a seguinte saída:

```
./Sucuri -s expressoes.txt
Estrutura de Dados 1 (2020/2) - EP2
Sucuri 1.0.0 (Apr  6 2021, 18:41:11)
[GCC 7.5.0] on Linux
Analisando a expressão: '2^3+1'
9
Analisando a expressão: '2^(3+1)'
16
Analisando a expressão: '1/2'
0
Analisando a expressão: '1.0/2'
0.500000
Analisando a expressão: '_1'
-1
Analisando a expressão: '3^2^2'
81
Analisando a expressão: '2*_3^4-2'
-164
Analisando a expressão: '_3^2'
-9
Analisando a expressão: '(_3)^2'
9
```

Visão geral do programa

O programa `Sucuri` lê linhas com expressões de um arquivo ou interativamente a partir do prompt. Cada linha possui uma expressão infixa supostamente correta. As linhas são percorridas e para cada linha o programa deve:

1. imprimir a expressão analisada (se a entrada estiver vindo de um arquivo);
2. criar uma fila com os itens léxicos da expressão: função `criaFilaObjetos` (`Lexer.c`);
3. percorrer a fila de itens substituindo no campo valor de cada objeto:
   - strings representando floats (categoria `FLOAT_STR`) por `double` (categoria `FLOAT`);
   - strings representando inteiros (categoria `INT_STR`) por `int` (categoria `INT`);
   - strings representando operador pela precedência do operador.

   A função `converteElementosFila` (`Fila.c`) é responsável por essas substituições.
4. percorrer a nova fila resultante e produzir uma nova fila que representa a expressão em notação pós-fixa. A função `infixaParaPosfixa` (`Posfixa.c`) faz essa tarefa;

5. percorrer a fila representando a expressão em notação pós-fixa e calcular o seu valor. A função `avalia` (`Avalia.c`) faz esse calculo;
6. imprimir o valor da expressão.

O que entregar

Você deve entregar, pelo **AVA**, um arquivo compactado contendo todos os códigos.

Data de entrega: até às 6h do dia 26/04/2021.

Observações:

1. Não mude a estrutura dos arquivos enviados. Você deve entender e completar o código. Se precisar adicionar alguma função auxiliar, adicione no arquivo de implementação (arquivo `.c`);
2. Preferencialmente, use o Linux (ou o CS50 IDE) para a implementação. No Windows, você pode usar o Code::Blocks para auxiliar na compilação do projeto;
3. Seu código deve ser compilado com as flags:
   ```
   -O0 -std=c11 -Wall -Werror -Wextra -Wno-sign-compare -Wno-unused-parameter -Wno-unused-variable -Wshadow
   ```
4. Códigos com erros de sintaxe (que não compilem) receberão nota 0;
5. Não mude o nome dos arquivos nem acrescente novos arquivos. A correção será feita usando o `Makefile` disponível no arquivo `Sucuri.zip`.
6. Código com vazamento de memória, valerá 70% da nota do EP;
7. Código que não segue o Guia de Estilo, valerá 90% da nota do EP;
8. **Em caso de plágio, será atribuído 0 a todos os envolvidos.**

Critérios de avaliação

A nota do EP se dará pela seguinte fórmula:

$$(1 - P) \times (M \times G \times N_{EP}),$$

onde,

- $P = \begin{cases} 1, & \text{se houve plágio;} \\ 0, & \text{caso contrário.} \end{cases}$
- $M = \begin{cases} 1.0, & \text{se o código não possui vazamento de memória;} \\ 0.7, & \text{caso contrário.} \end{cases}$
- $G = \begin{cases} 1.0, & \text{se seguiu o Guia de Estilo;} \\ 0.9, & \text{caso contrário.} \end{cases}$
- N_{EP}: Nota geral do EP, sendo $0.0 \leq N_{EP} \leq 10.0$;