Trabalho II de CI067 - Oficina de Computação
2º Semestre - 2018
Tema: Editor compactador de arquivos
Data de Entrega:

1. Introdução

O editor a ser implementado `edit([arquivo])` edita arquivos de texto ASCII. É um editor de linha interativo que lê linhas de comandos na entrada padrão e de acordo com comandos também fornecidos mostra resultados na saída padrão. O editor trabalha com um buffer que armazena as linhas e que, também seguindo a comandos, poderá ter seu conteúdo salvo em `arquivo`.

O buffer deverá ser organizado como uma sequência de linhas, numeradas a partir de 1. Estas linhas são automaticamente renumeradas à medida que ocorram inclusões ou remoções de linha. Importante: não existe um tamanho conhecido para cada linha. (*Portanto, use alocação dinâmica para armazenar cada linha no buffer).*

Os comandos podem ou não ser precedidos por um indicador de linha (exceto `q` que não admite nenhum número). Se o comando necessitar algum número específico de linhas e estes não forem devidamente fornecidos, o editor deverá informar ao usuário fornecendo o seguinte indicador de erro ou comando incorreto `?`.
Portanto será necessário realizar verificações para ver se os comandos podem ou devem ser executados para não haver perda de dados digitados caso o usuário se engane.

Utilizam-se as seguintes notações para um indicador de linha:
`n` número decimal que indica uma linha qualquer
`n1:n2` indica o conjunto de linhas entre `n1` e `n2`, incluindo estas.
Se nada for fornecido a última linha do arquivo será considerada, caso o buffer estiver vazio um número `n = 0` deve ser usado. Esta é uma convenção, mas não haverá linha 0 no buffer.

2. Comandos possíveis e formatos

Abaixo seguem os comandos e suas particularidades, bem como exemplos para o indicador de linha

****
insert: `[ indicadorLinha ] i texto`
Insere `texto` (dado por uma ou mais linhas fornecidas na entrada padrão e finalizado com dois pontos seguido de x (`:f <enter>`)) após posição dada por `indicadorLinha`. Veja que `indicadorLinha` está entre colchetes, portanto é opcional. Se nada for especificado, `texto` é adicionado após a última linha. A nova linha será a última linha do texto inserido.

```
i Adiciona esta frase após a última ou como primeira caso vazio.
:f
```

```
0 i Adiciona esta frase após a linha 0
ou seja como primeira e segunda linhas do arquivo.
:f
4 i Adiciona esta frase apos a linha 4.
& i        //  imprime ? Indicador de linha incorreto na saída padrão
i          //  imprime ? Texto requerido na saída padrão
```

No primeiro caso nenhum número de linha foi fornecido, então a linha é inserida no final. No segundo caso, são duas frases. A primeira será a primeira linha do arquivo, a segunda a segunda. No terceiro caso após a linha 4. No penúltimo caso foi fornecido um indicador de linha incorreto, então, um erro deverá ser retornado `? Indicador de linha incorreto` na saída padrão. No último caso não foi fornecido `texto`, então nada é feito e o editor retorna ao usuário um `? Texto requerido` na saída padrão. No início da edição, o buffer está vazio. Portanto apenas o primeiro e segundo comandos poderiam ser executados, os outros também retornariam `? Indicador de linha incorreto.` Se um indicador de linha do tipo `n1:n2` for fornecido a mensagem de erro também deverá ser fornecida. Lembre-se de fazer sempre estas verificações com os outros comandos também.

****

delete: `[ indicadorLinha ] d`
Remove o conteúdo das linhas do buffer indicado por `indicadorLinha.` As linhas que seguem o texto removido são automaticamente renumeradas.

```
4:5 d      // remove linhas 4 e 5.
4 d        // remove linha 4
d          // remove última linha se ela existir
4:& d      //  imprime ? Indicador de linha incorreto na saída padrão
0 d        //  imprime ? Indicador de linha incorreto na saída padrão
```

****

print: `[ indicadorLinha ] p`
Imprime na saída padrão o conteúdo das linhas do buffer indicado por `indicadorLinha.` É um comando para que o usuário saiba onde se encontra o indicador de linha.

```
p          // imprime  última linha se ela existir.
4:5 p      // imprime linhas 4 e 5
4 p        // imprime linha 4
#:  p      // imprime ? Indicador de linha incorreto na saída padrão
```

****

quit : q
Termina a execução do editor e não grava o conteúdo do arquivo. Não é obrigatório nesta versão avisar o usuário.

****

write: `[ indicadorLinha ] w [arquivo] [-c]`
Grava em `arquivo` o conteúdo do buffer indicado por `indicadorLinha` (a situação atual do buffer permanece inalterada). Se nada for especificado somente o conteúdo da última linha será armazenado. Se `arquivo` já existir seu conteúdo será substituído e não é necessário avisar o usuário.

```
w programa.c        // grava a última linha do buffer em programa.c se não vazio
                    // programa.c passa a ser o nome do arquivo sendo editado,
                    // descarta o nome fornecido como argumento para edit

1:4 w programa.c    // equivalente ao exemplo anterior, mas grava as linhas de 1 a 4.

w                   // Imprime ? Arquivo requerido na saída padrão,
                    // caso nenhum argumento tiver sido fornecido para edit ou
                    // grava no arquivo com o nome especificado no argumento de
                    // edit ou no comando write.
```

Importante: O programa `edit` poderá ou não ter um argumento na linha de comando que indica o nome do arquivo a ser editado, no qual se quer gravar o conteúdo de buffer. Portanto se nada for especificado para o comando `w`, o valor de `arquivo` será o mesmo especificado na linha de comando. No caso de nada também haver sido especificado na linha de comando, o comando retornará um `? Arquivo requerido.`
No caso de haver sido especificado o argumento e também um novo nome de arquivo no comando `w`, este último é utilizado e substituirá o nome dado como argumento.

Além disso, o comando `write` tem um parâmetro adicional `-c` que permite escrever o arquivo de maneira compactada, acrescentando no nome dado a extensão `.z.`

```
w programa.c -c     // grava a última linha do buffer em programa.c.z
                    // programa.c passa a ser o nome do arquivo sendo editado,
                    // descarta o nome fornecido como argumento para edit

1:4 w programa.c -c // equivalente ao exemplo anterior, gravando linhas de 1 a 4.

w -c                // Imprime ? Arquivo requerido na saída padrão,
                    // caso nenhum argumento tiver sido fornecido para edit ou
                    // grava no arquivo com o nome especificado no argumento de
                    // edit acrescentando a extensão .z e compactando
```

Para compactar um arquivo, deve-se utilizar a seguinte regra. Serão compactadas cadeias de 4 caracteres idênticos (e consecutivos) que serão substituídas por uma sequência codificada usando menos caracteres. A sequência de `x` caracteres é codificada com `~nx`, onde `n` é um caracter maiúsculo substituindo '`A`' significa uma repetição de `x`, '`B`' duas, e assim por diante. Considere 26 letras, e portanto sequências maiores que 26 devem ser quebradas em sequências menores. Por exemplo a sequência `aaacccccbb~` produz um arquivo de mesmo nome com extensao `.z`. contendo: `aaa~Ecbb~A~`. Note que o `~` é sempre codificado independentemente do número de vezes em que ele aparece. Os demais caracteres são codificados somente se aparecem 4 ou mais vezes.
****

read: `[indicadorLinha] r arquivo`
Insere conteúdo de `arquivo` na posição dada por `indicadorLinha`. Veja que `arquivo` deve existir e se não existir isso resultará em erro. Se nada for especificado o conteúdo será adicionado após a última linha. Se esta existir. Similar ao comando `insert`.

```
r data.txt      // Adiciona conteúdo de data.txt após a última linha
4 r entrada     // Adiciona conteúdo de entrada após a linha 4
& r entrada     // exibe ? Indicador de linha incorreto
r               // Imprime ? Arquivo requerido na saída padrão
```

Se o parâmetro `arquivo` possuir a extensão `.z` então será necessário expandir seu conteúdo antes da inserção. Deve-se assumir que seu conteúdo foi compactado pelo comando `write`. Para descompactar os arquivos, sequências da forma `~nc` são substituídas pelos caracteres repetidos que representam, de maneira que a saída seja igual ao texto antes da codificação. A ocorrência do caractere `~` na entrada significa que o caractere seguinte (que deve ser um caractere maiúsculo funciona como um contador de repetições; '`A`' indica uma repetição, '`B`' indica duas e assim sucessivamente de '`A`' até '`Z`'. Por exemplo, a linha contendo `a~Db~A~~Db` deverá ser expandida e conterá `abbbb~bbbb` ao final.

3. Exemplos de usos

Abaixo são fornecidos alguns exemplos de uso do `edit`. Utilize-os para testar o seu programa antes de entregar.
Uma opção, utilizada abaixo, para tornar mais legível é apresentar tudo o que buffer contiver a cada comando executado.

Exemplo 1: Como é fornecido o nome programa.c o comando `write` não requer nome de arquivo.

```
> edit programa.c
0 i #include <stdio.h>
main( )
{
    printf("primeiro programa testando o ccccccc~ccccdiiii");
}
:f < enter>  para finalizar o comando insert

-------------------------------------------------------------
 1 #include <stdio.h>
 2 main( )
 3 {
 4    printf("primeiro programa testando o ccccccc~ccccdiiii");
 5 }
-------------------------------------------------------------

2 d

-------------------------------------------------------------
 1 #include <stdio.h>
 2 {
 3    printf("primeiro programa testando o ccccccc~ccccdiiii");
 4 }
-------------------------------------------------------------

1 i int main( )
:f

-------------------------------------------------------------
 1 #include <stdio.h>
 2 int main( )
 3 {
 4    printf("primeiro programa testando o ccccccc~ccccdiiii");
 5 }
-------------------------------------------------------------

1-5 w

-------------------------------------------------------------
 1 #include <stdio.h>
 2 int main( )
 3 {
 4    printf("primeiro programa testando o ccccccc~ccccdiiii");
 5 }
-------------------------------------------------------------
```

```
q
>
> more programa.c
#include <stdio.h>
int main( )
{
  printf("primeiro programa testando o ccccccc~ccccdiiii");
}
>
```

Exemplo 2: Como não foi fornecido nenhum nome de arquivo no comando `edit`, o comando `write` requer nome de arquivo. O arquivo será compactado pois o parâmetro `c` foi utilizado, gerando o arquivo `programa.c.z`. O arquivo programa.c fica inalterado.

```
> edit
r programa.c

----------------------------------------------------------------
 1 #include <stdio.h>
 2 int main( )
 3 {
 4    printf("primeiro programa testando o ccccccc~ccccdiiii");
 5 }
----------------------------------------------------------------

1-5 w

? Arquivo requerido

1-5 w programa.c -c

----------------------------------------------------------------
 1 #include <stdio.h>
 2 int main( )
 3 {
 4    printf("primeiro programa");
 5 }
----------------------------------------------------------------

q

>
> more programa.c.z
#include <stdio.h>
int main( )
{
  printf("primeiro programa testando o ~Gc~A~~Dcd~Di");
}
>
```

Exemplo 3: Como neste caso foi fornecido o nome de arquivo `teste.c` como argumento de `edit`, o comando `write` não requer nome de arquivo. O arquivo lido no comando `read` tem extensão `.z` e será automaticamente expandido.

```
>  edit teste.c
r programa.c.z

----------------------------------------------------------------
 1 #include <stdio.h>
 2 int main( )
 3 {
 4    printf("primeiro programa testando o ccccccc~ccccdiiii");
 5 }
----------------------------------------------------------------

1-4 p

----------------------------------------------------------------
 1 #include <stdio.h>
 2 int main( )
 3 {
 4    printf("primeiro programa testando o ccccccc~ccccdiiii");
----------------------------------------------------------------

9 d

? Identificador de linha incorreto

1-2 w

----------------------------------------------------------------
 1 #include <stdio.h>
 2 int main( )

----------------------------------------------------------------

q

> more teste.c
#include <stdio.h>
int main( )
```

4. Condições de entrega e avaliação

Você deve criar um arquivo com extensão .tar e utilizar o programa entrega, assim como os exercícios entregues em sala, com o comando abaixo.
> /home/html/inf/silvia/entrega/bin/entrega oficinac 21 arquivo.tar

Os arquivos devem ser entregues até às 23:59 de domingo, dia 04/11. Trabalhos entregues depois do dia 04 terão 10 pontos descontados por dia de atraso. A data limite de entrega é dia 07/11.

O programa entregue será apresentado pelo próprio aluno em sala de aula no dia 08/11. Durante a apresentação serão feitas perguntas sobre a implementação.

O trabalho vale 100 pontos. A avaliação seguirá os seguintes critérios e valores.

1) funcionalidade (80): relacionada ao correto funcionamento de todos os 8 comandos: edit, insert, delete, print, write e read, com as opções de comprimir e expandir arquivo.

2) a estruturação e organização do código (10) e a legibilidade (5) conforme material passado em aula. Utilize funções, declare suas funções, comentários adequados, arquivos .h e .c

3) arquivo makefile é obrigatório (5) pelo menos com uma variável.

4) defesa do trabalho (5): respostas corretas a perguntas feitas no dia da apresentação.

Faça funções de leitura e impressão de linhas, de reconhecimento de indicadores de linha, e de comandos que serão utilizadas por todos os comandos. Declare nos .h somente macros, diretivas, tipos e protótipos de funções. Não inclua código C nestes arquivos. Não utilize variáveis globais.
Cada um dos comandos pode ser um programa C que pode ser testado a parte, por isto utilize o *make*, quando um comando estiver implementado e funcionando, você não precisará recompilá-lo.

Caso o programa enviado não compile, a nota será 0. Teste muito bem o seu programa e certifique-se que ele está funcionando corretamente. Preferivelmente guarde versões dele, pois se acontecer algum imprevisto, uma versão anterior poderá ser utilizada.
Caso o programa compile, mas não execute, o código será avaliado pela sua legibilidade e organização. Retire todos os *warnings.*
Caso o programa execute, porém falhe durante a execução por um erro de programação, então o programa será avaliado individualmente e será dada uma nota relativa aos requisitos atendidos.
Durante a apresentação do trabalho serão feitas perguntas sobre como o aluno implementou o programa. Caso o aluno não saiba responder, serão descontados pontos por pergunta errada.

Quaisquer funcionalidades extras que não tenham sido pedidas no enunciado são opcionais e não contribuem para a nota.