MAC0122 – Princípios de Desenvolvimento de Algoritmos

Poli/Elétrica, Opção Sistemas Eletrônicos — Primeiro Semestre de 2010

Terceiro Exercício-Programa — Data de entrega: até **27 de junho de 2010**

Processador de Macros e de Inclusões

Neste exercício programa você implementará uma versão simplificada do pré-processador da linguagem C. Seu programa, denominado `processa`, receberá como entrada um arquivo de texto e produzirá como saída outro arquivo de texto. Ele atuará como um processador de macros e de inclusões de arquivos, aceitando as seguintes diretivas:

- Inclusão de arquivo:

  `#include "`*`nome_arq`*`"`

  `#include <`*`nome_arq`*`>`

  Cada linha do arquivo de entrada que tiver uma dessas formas será substituída, no arquivo de saída, pelo conteúdo do arquivo cujo nome é *`nome_arq`*.

- Definição de macro:

  `#define` *`nome_da_macro expansao_da_macro`*

  Cada linha do arquivo de entrada que tiver essa forma aparecerá no arquivo de saída como uma linha vazia, mas terá o efeito de associar o *`nome_da_macro`* à expansão especificada. Ocorrências subsequentes do *`nome_da_macro`* (no arquivo de entrada) serão substituídas (no arquivo de saída) pela *`expansao_da_macro`*.

  O *`nome_da_macro`* deve ser um identificador, isto é, uma sequência de letras e dígitos começando com letra. Os caracteres `'_'` e `'$'` valem como letras. Já a *`expansao_da_macro`* pode ser um texto qualquer.

- Remoção de macro previamente definida:

  `#undef` *`nome_da_macro`*

  Cada linha do arquivo de entrada que tiver essa forma aparecerá no arquivo de saída como uma linha vazia, mas terá o efeito de remover a associação entre o *`nome_da_macro`* e a correspondente expansão. Ocorrências subsequentes do *`nome_da_macro`* serão copiadas diretamente para o arquivo de saída.

Argumentos na linha de comando

O programa `processa` deve aceitar os seguintes argumentos na linha de comando:

`./processa [`*`arq_entrada`*`] [`*`arq_saida`*`]`

Os argumentos aceitos pelo programa estes significados:

- O argumento *`arq_entrada`* é o nome do arquivo de entrada.

- O argumento *arq_saida* é o nome do arquivo de saída.

Exemplo:

```
./cliente entrada.txt saida.txt
```

O tratamento dado a cada argumento é determinado pela posição do argumento na linha de comando. O primeiro argumento é o *arq_entrada* e o segundo é o *arq_saida*. Pode ocorrer omissão de argumentos:

- Se o primeiro argumento estiver presente e o segundo for omitido, o programa considerará que o *arq_entrada* é o primeiro (e único) argumento e que o *arq_saida* é a saída padrão.
- Se os dois argumentos forem omitidos, o programa considerará que o *arq_entrada* é a entrada padrão e que o *arq_saida* é a saída padrão.

Visão geral da implementação

O programa `processa` mantém uma tabela que associa nomes a expansões de macros. Este pseudocódigo dá uma visão aproximada do laço principal do programa `processa`:

```
enquanto (le_proximo_item(entrada, item) != FIM_DE_ARQUIVO) {
    se (item é "#include")
        trata a inclusao de arquivo
    senao se (item é "#define")
        insere na tabela de macros um novo par <nome, expansao>
    senao se (item é "#undef")
        remove da tabela de macros um par <nome, expansao>
    senao se (item é identificador) {
        expansao_da_macro = busca_na_tabela_de_macros(item)
        se (expansao_da_macro != NULL)
            expande a macro cujo nome é item
        senao
            copia o item para a saída
    }
    senao
        copia o item para a saída
}
```

Leitura de itens léxicos

Seu programa deve conter uma função que faz a leitura do próximo item léxico de um arquivo de entrada. O laço principal do programa chama essa função, cujo protótipo deve ser

```
int le_item(char item[], int lim, FILE *entrada);
```

A função `le_item` coloca no vetor `item` uma *string* com o próximo item léxico do arquivo de `entrada` especificado como parâmetro. O valor devolvido pela função é o comprimento dessa *string*, ou -1 caso não exista o próximo item (fim do arquivo de entrada).

O parâmetro `lim` especifica o número máximo de caracteres que pode ser colocado no vetor `item`. A função `le_item` colocará nesse vetor uma *string* de comprimento menor ou igual a `lim-1`.

Os itens lidos pela função `le_item` têm uma das seguintes formas:

- O item é uma "palavra", isto é, uma *string* contendo letras, dígitos, e caracteres `'_'`, `'$'` e `'#'`.
- O item é um caractere especial. Neste caso a função coloca no vetor `item` uma *string* de comprimento 1, cujo único caractere não pode ser nenhum dos que podem aparecer numa palavra.

Note que os identificadores (os possíveis nomes de macros) são um subconjunto das palavras: um identificador é uma palavra que não contém o caractere `'#'` e não começa com dígito.

Devolução de caracteres para releitura

Quando estiver implementando a função `le_item`, você terá o seguinte problema: Como saber que a palavra atual acabou? A função `le_item` só perceberá que uma palavra acabou quando ler um caractere que não pode fazer parte da palavra. O que fazer com esse caractere?

Uma solução elegante para esse problema é "empurrar de volta" para a entrada o caractere lido a mais, de modo que ele seja lido novamente quando o programa quiser obtê-lo. Para viabilizar essa solução, um dos módulos do seu programa deve implementar um par de funções com os seguintes protótipos:

```
int le_caractere(FILE *entrada);
void empurra_caractere(char c);
```

Esse par de funções usa uma pilha de caracteres empurrados de volta para a entrada. A função `empurra_caractere` recebe um caractere `c` e o coloca na pilha de caracteres. A função `le_caractere` verifica se há algum caractere nessa pilha. Em caso afirmativo, ela devolve um caractere retirado da pilha. (Esse é o último caractere que foi "empurrado de volta para a entrada".) Em caso negativo, ela devolve o resultado de uma chamada `fgetc(entrada)`. Note que `le_caractere` devolverá `EOF` quando for atingido o final do arquivo de `entrada` (pois `fgetc` faz isso).

Implemente a pilha de caracteres como um vetor compartilhado pelas funções `le_caractere` e `empurra_caractere`. Crie o módulo `leitura.c`, que contém as variáveis usadas pela pilha (declaradas como globais, porém acessíveis somente de dentro do módulo `leitura.c`) e as funções `le_caractere` e `empurra_caractere`.

```
/* arquivo leitura.c */
#include "leitura.h" /* contém os protótipos das funções públicas do modulo */

#define TAM_PILHA 1000000

static char pilha[TAM_PILHA];
static int topo = 0;

int le_caractere(FILE *entrada)
{
    ...
}
```

```
void empurra_caractere(char c)
{
    ...
}
```

Para que a estratégia de "empurrar caracteres de volta para a entrada" funcione, toda leitura deve ser feita por meio de uma chamada à função `le_caractere`. A função `le_item` nunca efetuará uma chamada direta a `fgetc` ou a alguma outra função de `<stdio.h>` que faça leitura de dados. Ela chamará `le_caractere` sempre que quiser ler o próximo caractere, e chamará `empurra_caractere` quando quiser se livrar de um caractere lido a mais.

A função `processa` e o tratamento de inclusões

Seu programa deve conter uma função com o seguinte protótipo:

```
void processa(FILE *entrada, FILE *saida);
```

Essa é a função central do programa `processa`. É nela que está o laço principal do programa. A função `main` cuida apenas de (quando for o caso) abrir os arquivos de entrada e de saída e em seguida efetua uma chamada a `processa`. Todo o processamento de macros e de inclusões de arquivos ocorre dentro dessa chamada. Ao final da chamada a `processa`, a função `main` (quando for o caso) precisará apenas fechar os arquivos de entrada e de saída.

Use recursão para tratar inclusões de arquivos (linhas da forma `#include ...`) de modo simples e elegante. Faça a função `processa` chamar a si mesma quando encontrar uma linha com `#include`. Numa chamada recursiva a `processa`, o parâmetro `entrada` deve estar associado ao arquivo especificado no `#include`. (A função `processa` precisará abrir esse arquivo. Não se esqueça de fazer com que ela o feche na ocasião adequada!)

A tabela de macros

A tabela de macros deve ser implementada como uma tabela de hash que associa cada nome de macro à expansão correspondente. Os nomes das macros são as chaves de busca na tabela de hash.

Crie o módulo `tabela.c`, cuja interface (`tabela.h`) consiste das seguintes funções:

```
/* arquivo tabela.h */
void inicia(int tamanho);
int insere(char *nome, char *expansao);
int remove(char *nome);
char *busca(char *nome);
```

O módulo `tabela.c` implementa uma tabela de hash como um vetor de listas encadeadas sem cabeça. Cada elemento desse vetor é um ponteiro cujo valor é o endereço da primeira célula da lista encadeada correspondente, ou `NULL` (caso essa lista seja vazia).

A função `inicia` faz a inicialização da tabela de hash. Ela recebe como parâmetro o tamanho do vetor de ponteiros, que deve ser alocado dinamicamente.

A função `insere` tenta adicionar à tabela um novo par (`nome`, `expansao`). Ela devolve 1 se a inserção foi bem sucedida, ou 0 se a inserção não foi efetuada porque o `nome` especificado já estava presente na tabela.

A função `remove` tenta remover da tabela o par com o `nome` especificado. Ela devolve 1 se a remoção foi bem sucedida, ou 0 se a remoção não foi efetuada porque o `nome` especificado não estava presente na tabela.

A função `busca` faz uma busca pelo `nome` especificado e devolve a expansão correspondente, ou `NULL` caso o `nome` não apareça na tabela.

Processamento de macros

O tratamento de `#define` e de `#undef` consiste essencialmente de chamadas às funções `insere` e `remove`. Já o tratamento de cada ocorrência de um identificador (um possível nome de macro) começa com uma chamada à função `busca`. Se o identificador for um nome de macro, ele deve ser substituído pela expansão correspondente. Aqui o problema é expandir a macro levando em conta que o texto da expansão pode conter outros nomes de macros, os quais também devem ser expandidos. O seguinte arquivo de entrada ilustra esse caso:

```
#define PI 3.14
#define R  42
#define AREA (PI*R*R)

... AREA ...
```

Nesse exemplo, a expansão da macro `AREA` é o texto "`(PI*R*R)`", que contém três ocorrencias de nomes de macros (uma ocorrência de `PI` e duas de `R`). Essas ocorrências também precisam ser expandidas!

Note que a ordem dos `#define`s não deve ser relevante. O arquivo de saída não deve mudar se o arquivo de entrada acima for reescrito da seguinte maneira:

```
#define AREA (PI*R*R)
#define PI 3.14
#define R  42

... AREA ...
```

Uma maneira simples e elegante de lidar com o problema da expansão de uma macro é usar o recurso de "empurrar caracteres para a entrada". Nessa abordagem, o tratamento da macro `AREA` "empurraria para a entrada" a sequência de caracteres "`)R*R*IP(`", que corresponde à expansão da macro em ordem reversa. Como os caracteres empurrados são colocados numa pilha, as chamadas subsequentes a `le_caractere` devolverão os caracteres da expansão da macro na ordem correta, como se eles viessem do arquivo de entrada.

Inclua no módulo `leitura.c` uma função com o seguinte protótipo:

```
void empurra_texto(char *texto);
```

Essa função recebe uma string `texto` e faz uma série de chamadas a `empurra_caractere` de modo a empurrar para a entrada os caracteres do `texto` em ordem reversa. Para expandir uma macro com um certo *`nome`*, a função processa precisa apenas obter da tabela de a *`expansao`* correspondente e executar `empurra_texto(`*`expansao`*`)`.

O problema dos laços perpétuos

> **Sugestão:** Na versão inicial do seu programa implemente o que foi especificado até agora, sem se preocupar com o problema dos laços perpétuos. Ataque esse problema só depois que a versão inicial do programa estiver funcionando!

Se o programa `processa` for implementado da maneira descrita nas seções anteriores deste enunciado, o seguinte arquivo de entrada o fará entrar num laço perpétuo:

```
#define XIS XIS
XIS
```

Qual o problema com esse arquivo de entrada? Quando o programa `processa` encontra o `#define`, ele coloca na tabela de macros uma macro cujo nome é `XIS` e cuja expansão também é `XIS`. Quando ele encontra a ocorrência de `XIS` na segunda linha, ele "empurra para a entrada" a expansão correspondente, que também é `XIS`. Portanto o programa encontrará uma nova ocorrência de `XIS`... Laço perpétuo!

O mesmo problema se manifesta com arquivos de entrada com "definições circulares" mais complexas, como por exemplo este:

```
#define A (B+C)
#define B (10*D*E)
#define D (F-G)
#define F (A*A/2)
A
```

O que o pré-processador da linguagem C faz nesses casos? Quando uma expansão de macro contiver alguma ocorrência de um nome de uma macro que está sendo expandida, o pré-processador do C não expandirá a ocorrência do nome. No primeiro dos dois exemplos acima, a ocorrência de `XIS` na segunda linha do arquivo de entrada será substituída por sua expansão, mas a ocorrência de `XIS` dentro da expansão de `XIS` será copiada diretamente para o arquivo de saída. Ou seja, a segunda linha do arquivo de entrada aparecerá no arquivo de saída também com um `XIS`.

No segundo dos exemplos acima, a ocorrência de `A` na quinta linha do arquivo de entrada será substituída por sua expansão `(B+C)`, a ocorrência de `B` nessa expansão será substituída por `(10*D*E)`, a ocorrência de `D` dentro da expansão de `B` será substituída por `(F-G)`, a ocorrência de `F` dentro da expansão de `D` será substituída por `(A*A/2)`, mas as ocorrências de `A` dentro da expansão de `F` não serão expandidas. Elas serão copiadas diretamente para o arquivo de saída, pois ainda está em curso a expansão da ocorrência de `A` na quinta linha do arquivo de entrada! A quinta linha do arquivo de saída será

```
((10*((A*A/2)-G)*E)+C)
```

Queremos que o programa `processa` também tenha esse comportamento. Para isso, ele precisará "saber" que macros estão sendo expandidas a cada momento.

A pilha de macros ativas

Faça seu programa manter uma pilha de macros ativas, isto é, uma pilha de *strings* contendo os nomes das macros que estão sendo expandidas no momento. Essa pilha pode ser implementada como um vetor de *strings*. Além das operações usuais `empilha` e `desempilha`, a pilha de macros ativas deve oferecer também uma operação de busca, que verifica se uma dada *string* aparece em alguma posição da pilha.

Tendo a pilha de macros ativas, fica muito fácil implementar o comportamento do pré-processador da linguagem C. Antes de expandir uma macro (ou seja, antes de fazer uma chamada a `empurra_texto` passando como parâmetro a expansão da macro), verifique se a macro já está ativa. Em outras palavras, verifique se o nome da macro aparece em alguma posição da pilha de macros ativas. Se a macro já estiver ativa, não a expanda. Simplesmente copie o nome da macro para o arquivo de saída.

Resta agora o problema de gerenciar a pilha de macros ativas. Em que momento o nome de uma macro deve ser empilhado? Em que momento ele deve ser desempilhado?

A primeira questão é a mais fácil. Sempre que você chamar `empurra_texto` passando como parâmetro a expansão de uma macro, empilhe o nome dessa macro. É conveniente criar uma função `expande_macro` para fazer as duas coisas:

```
void expande_macro(char *nome, char *expansao)
{
    empilha(nome);
    empurra_texto(expansao);
}
```

A segunda questão é um pouco mais difícil. Como saber que acabou a expansão de uma macro? O jeito é empurrar para a entrada uma marca que indique o final da expansão. Essa marca deve ser um caractere que não apareça em nenhum arquivo de entrada. Lá pelas tantas ocorrerá uma chamada à função `le_item` que devolveria a marca de fim de expansão (um caractere especial). Em vez de devolver essa marca (que certamente não tem interesse para o chamador de `le_item`), a função `le_item` deve desempilhar o nome de macro que estiver no topo da pilha de macros ativas e devolver o item que vier depois da marca de fim de expansão.

Use o caractere nulo (`'\0'`) como marca de fim de expansão. Esta versão melhorada da função `expande_macro` cuida de empurrar essa marca para a entrada no momento adequado:

```
void expande_macro(char *nome, char *expansao)
{
    empilha(nome);
    empurra_caractere('\0'); /* marca o final da expansão */
    empurra_texto(expansao);
}
```

Observações

- O exercício-programa é estritamente individual. Veja nesta URL a política do Departamento de Ciência da Computação para casos de plágio ou cola: `http://www.ime.usp.br/webadmin/dcc/grad/sobreplagio`.
- Exercícios atrasados NÃO serão aceitos.
- Exercícios com erros de sintaxe (ou seja, erros de compilação) receberão nota ZERO. Seu programa deve ser compilável sem erros ou warnings, da maneira especificada abaixo (que usa o compilador num modo em que quase todos os warnings são emitidos).

- Para compilar seu programa, use o `gcc` ou o `Code::Blocks` (que na verdade chama o `gcc` para fazer a compilação). Caso você use diretamente o `gcc`, passe ao compilador (na linha de comando) as seguintes opções:

  `-Wall -ansi -pedantic -O2 -U_FORTIFY_SOURCE`

  Caso você use o `Code::Blocks`, entre em *Settings* → *Compiler and debugger...* → *Compiler settings* → *Compiler Flags*, selecione as quatro opcões correspondentes a `-Wall`, `-ansi`, `-pedantic` e `-O2`, e clique em *OK*. Entre também em *Settings* → *Compiler and debugger...* → *Compiler settings* → *Other options*, digite `-U_FORTIFY_SOURCE` na caixa de texto *Other options* e clique em *OK*.

- Seu programa deve estar bem indentado, documentado e organizado. A identação deve deixar clara a estrutura de subordinação dos comandos. Os comentários devem ser esclarecedores. Toda função deve ser precedida de um comentário que diz o que a função faz. As funções devem ser razoavelmente pequenas, na medida do possível, e cada uma delas deve ter um propósito bem definido. A saída do programa deve ser clara. A avaliação levará em conta todas essas questões! Uma apresentação ruim, ou a falta de clareza do programa ou da saída do programa, poderá prejudicar sua nota.

- O programa deve ser entregue por meio do sistema Paca/Moodle.

- Entregue um arquivo tar.gz ou zip que satisfaça os seguintes requisitos:

  - O nome do arquivo deve ser da forma `ep3-`<*seu-número-USP*>`.tar.gz` ou `ep3-`<*seu-número-USP*>`.zip`. Por exemplo: `ep3-12345678.zip`.

  - O desempacotamento do arquivo tar.gz ou zip deve produzir um diretório com o mesmo nome do arquivo, menos o sufixo `.tar.gz` ou `.zip`. (Exemplo: `ep3-12345678`.) Todos os arquivos desempacotados devem estar dentro desse diretório.

  - O diretório desempacotado deve conter:

    * Os arquivos `leitura.h` e `leitura.c`, contendo respectivamente a interface e a implementação do módulo de leitura.
    * Os arquivos `tabela.h` e `tabela.c`, contendo respectivamente a interface e a implementação da tabela de macros (uma tabela de hash).
    * Os arquivos `pilha.h` e `pilha.c`, contendo respectivamente a interface e a implementação da pilha de macros ativas (uma pilha de *strings*).
    * Um arquivo `processa.c`, contendo as funções `main`, `processa`, `le_item` e `expande_macro`.
    * Quaisquer outros arquivos `.h` ou `.c` que você utilizar para funções auxiliares compiladas separadamente. Por exemplo, se você tiver um módulo compilado separadamente contendo apenas a função `mallocX`, então você deverá entregar os arquivos `mallocx.h` e `mallocx.c`.

  - Todos os seus arquivos `.h` ou `.c` devem ter um cabeçalho como o seguinte:

```
/*****************************************************************/
/* Aluno: Fulano de Tal                                          */
/* Número USP: 12345678                                          */
/* Curso: ...                                                    */
/* Exercicio-Programa 3 -- Processador de Macros e de Inclusões */
/* MAC0122 -- 2010 -- POLI/Eletrica (PSI), -- Prof. Reverbel     */
/* Compilador: ... (gcc ou Code::Blocks) versão ...              */
/* Sistema Operacional: ...                                      */
/*****************************************************************/
```

- Enquanto o prazo de entrega não expirar, você poderá entregar várias versões do mesmo exercício-programa. Apenas a última versão entregue será guardada pelo sistema. Encerrado o prazo, o sistema não aceitará mais a entrega de exercícios-programa. Não deixe para entregar seu exercício na última hora!

- Guarde uma cópia do seu exercício-programa pelo menos até o final do semestre.

Bom trabalho!