Laboratorio di Informatica L-A
Prova d'Esame 3.1 – 04 Aprile 2007

Prima di cominciare: si scarichi il file **`StartKit3.1.zip`** contenente i file di testo necessari.
Avvertenze per la consegna: nominare i file sorgenti come richiesto durante il testo del compito, apporre all'inizio di ogni file sorgente un commento contenente i propri dati (**cognome, nome, numero di matricola**) e il **numero** della prova d'esame. Al termine, **consegnare tutti i file sorgenti** ed i file contenuti nello StartKit.
Rispettare **ALLA LETTERA** le specifiche, in particolare inserire le funzioni nei file specificati fra parentesi dopo il nome della funzione. Chi non rispetta le specifiche sarà opportunamente penalizzato. **NON SARANNO CORRETTI** gli elaborati che presenteranno un numero "non affrontabile" di errori di compilazione.
Consiglio: per verificare l'assenza di *warnings*, effettuare di tanto in tanto un *Rebuild All*.

Si scriva un'applicazione in grado di leggere un file contenente un testo qualsiasi (nello **StartKit** sono presenti due file da utilizzare) e di effettuare il conteggio delle parole presenti; in particolare, per ogni parola incontrata, è necessario memorizzare il numero di occorrenze. Al termine dell'elaborazione, l'applicazione deve scrivere su file di testo le parole individuate e la loro occorrenza (ovviamente le parole non vanno ripetute).
L'applicazione deve essere strutturata su due livelli: il livello più basso è costituito da un *tokenizer* che legge il file carattere per carattere, "filtra" i caratteri non significativi (punteggiatura, simboli, ecc.) e restituisce le singole parole di cui è composto il file stesso; al livello superiore c'è l'algoritmo di conteggio delle parole che utilizza i servizi esposti dal *tokenizer* e che memorizza i dati in una lista ad inserimento ordinato (secondo l'ordine alfabetico). Nello **StartKit** si trovano alcuni file di testo di esempio ed il file .h contenente la definizione della stringa con i caratteri di separazione da utilizzare.

Esercizio 1 – Tokenizer (tokenizer.h/tokenizer.c)
Il *tokenizer* deve essere progettato come ADT. Lo stato del *tokenizer* è rappresentato da un un file (**`FILE*`**) e una stringa (max 50 caratteri) che contiene tutti i caratteri da considerare come separatori, quindi non costituenti un *token*. La funzione **`createTokenizer`** prende in ingresso il nome del file di testo da scandire e la stringa contenente i caratteri di separazione e restituisce un puntatore alla struttura che deve essere allocata dinamicamente; nel caso in cui si verifichi un errore nell'apertura del file occorre restituire **`NULL`**. La funzione **`destroyTokenizer`** prende in ingresso un *tokenizer*, chiude il file contenuto nella struttura e dealloca la struttura stessa. La funzione **`getNextToken`** prende in ingresso un *tokenizer* e un *array* di caratteri (già allocato) dove inserire il *token* letto; tale funzione restituisce *vero* se il token è stato letto correttamente, *falso* (nessun *token* letto) altrimenti.
Suggerimenti
Si legga il file carattere per carattere; considerando che la funzione **`getNextToken`** verrà invocata più volte di seguito, prima si devono ignorare eventuali caratteri di separazione (si consiglia di utilizzare le funzioni sulle stringhe per sapere se il carattere corrente letto dal *file* è un carattere di separazione o meno) poi si deve comporre il *token* interrompendo la lettura non appena si incontra un carattere di separazione. Ovviamente, il carattere **`EOF`** interrompe in ogni caso la lettura. Prima di passare all'esercizio successivo si consiglia la verifica del corretto funzionamento del *tokenizer*.

Esercizio 2 – Word (word.h/word.c)
L'astrazione *Word* rappresenta semplicemente una stringa allocata dinamicamente. Si dichiari, quindi, il tipo **`Word`** (tramite una **`typedef`**) in modo che sia mappato su un puntatore a carattere. La funzione **`createWord`** prende come parametro un *array* di caratteri, e restituisce un puntatore ad una stringa allocata dinamicamente (la dimensione deve essere

quella minima necessaria a contenere la stringa contenuta nel parametro ricevuto) in cui va copiato il valore ricevuto. La funzione **`destroyWord`** prende come parametro un oggetto di tipo **`Word`** e dealloca la memoria corrispondente.

Esercizio 3 – Lista di parole (wordlist.h/wordlist.c)
Si creino le strutture dati adatte a creare una lista di parole; la parola deve essere modellata utilizzando il tipo **`Word`** (per usare meno memoria possibile) e, accanto ad ogni parola, deve essere memorizzato anche un intero che rappresenta un conteggio (si veda più avanti per il significato del conteggio). La funzione **`addWord`** prende come parametri una stringa (*array* di caratteri) ed una lista e restituisce una lista; le elaborazioni compiute dalla funzione sono le seguenti:

- se la stringa ricevuta non è presente nella lista, la si aggiunge e si pone ad uno il conteggio corrispondente;
- se la stringa ricevuta è presente nella lista, si incrementa il conteggio corrispondente;

in ogni caso viene restituita la lista modificata. All'atto dell'inserimento, si usi la funzione **`createWord`** precedentemente creata per ottenere una **`Word`** da una stringa. L'inserimento deve avvenire in modo ordinato: le parole contenute nella lista devono risultare lessicograficamente ordinate in modo crescente – questo consente di effettuare la ricerca in maniera un po' più efficiente. Il metodo **`destroyWordList`** prende come parametro una lista di parole e la dealloca completamente (attenzione: devono essere deallocate anche le singole **`Word`**). Prima di passare all'esercizio successivo si consiglia la verifica del corretto funzionamento della lista di parole.

Esercizio 4 – Conteggio e salvataggio (wordlist.h/wordlist.c)
Si realizzi una funzione di nome **`wordCounter`** che prenda come parametri il nome di un *file*, una stringa contenente i caratteri da considerare di separazione ed un riferimento ad una lista di parole. Per realizzare il conteggio delle occorrenze delle parole nel testo, occorre innanzitutto creare un *tokenizer* (alla funzione di costruzione vanno passati il nome del file e i caratteri di separazione), se la costruzione non è avvenuta correttamente **`wordCounter`** deve restituire *errore*, se la costruzione è avvenuta con successo occorre leggere tutte le parole contenute nel file (**`getNextToken`** – passare alla funzione una stringa di max 80 caratteri), inserirle una ad una in una lista di parole (**`addWord`**) e, al termine della lettura, distruggere il *tokenizer*, assegnare la lista creata al riferimento ricevuto come parametro e restituire *successo*. Si realizzi una funzione di nome **`saveWordList`** che prenda come parametri un file (**`FILE*`**) e una lista di parole e che scriva sul file le parole contenute nella lista ed i relativi conteggi. Un esempio di file di uscita è il file **`Results.txt`** all'interno dello **StartKit**.

Esercizio 5 – Il main (main.c)
Il **`main`** deve contenere tutto il codice necessario per far funzionare l'applicazione. In particolare deve contenere un'invocazione a **`wordCounter`** (come stringa di caratteri di separazione, usare quella contenuta nello **StartKit**) su uno dei file contenuti nello **StartKit** (**`TextFile.txt`**, **`ShortTextFile.txt`**). Nel caso l'invocazione di cui sopra abbia successo, aprire un file di testo e invocare la funzione **`saveWordList`** passando il file appena aperto e la lista di parole ottenuta da **`wordCounter`**. Terminato il salvataggio, chiudere il file, distruggere la lista di parole e terminare l'applicazione. Nel caso in cui l'invocazione a **`wordCounter`** termini con errore, stampare un messaggio e terminare l'applicazione.
Per un confronto, il file **`Results.txt`** è ottenuto partendo dal file **`TextFile.txt`**.