$Id: asg4c-spellchk-hash.mm,v 1.4 2012-11-16 21:24:29-08 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps012b-wm/Assignments/asg5c-spellchk-hash
URL: http://www2.ucsc.edu/courses/cmps012b-wm/:/Assignments/asg5c-spellchk-hash

1. Overview

In this assignment you will implement a spelling checker that uses a hash table to look up words in a dictionary. Collision resolution will be done by linear probing rather than chaining. A scanner generated by **flex** will be used to extract words form the files to be checked.

2. Program specification

We present the program in the form of a Unix **man**(1) page.

NAME

spellchk — spell check some files based on dictionary words

SYNOPSIS

spellchk [**-nxy**] [**-d** *dictionary*] [**-@** *debugflags*] [*filename* ...]

DESCRIPTION

This program examines files for correct spell checking. Some number of dictionaries are read in, including the default dictionary, plus any other auxiliary dictionaries. then each file is read, and a report of any incorrectly spelled words is made.

OPTIONS

Options are scanned using **getopt**(3c), and are subject to its restrictions and conventions. It is an error if no dictionaries are specified.

-d *dictionary*
The specified dictionary is loaded and used in addition to the default dictionary. This is optional unless **-n** is used.

-n The default special dictionary is excluded and only explicitly specified dictionaries are used.

-x Debug statistics about the hash table are dumped. If the **-x** option is given more than once, the entire hash table is dumped. The files to be spell checked are ignored if this option is specified.

-y Turns on the scanner's debug flag.

-@ *debugflags*
Turns on debugging flags for the **DEBUGF** macro. The option **-@@** turns on all debug flags.

OPERANDS

Each operand is the name of file whose words are to be checked for spelling errors. A word is any sequence of letters and digits, possibly with the characters ampersand (**&**), apostrophe (**'**), hyphen (**-**), or period (**.**) embedded with the word. If a filename is specified as a minus sign (**-**), it causes **stdin** it be read at that point. If no filenames are specified, **stdin** is spell checked.

EXIT STATUS

0 No errors nor misspelled words were detected.

1 One or more misspelled words were detected, but there were no errors.

2 One or more errors were detected and error messages were printed.

FILES

/afs/cats.ucsc.edu/courses/cmps012b-wm/usr/dict/words
Contains the default dictionary.

BUGS

Standard spell-checking algorithms for variations on words as to number and tense are not performed. In any case, poems like the one in Figure 1 are likely to confuse most spelling checkers.

Eye halve a spelling chequer
It came with my pea sea
It plainly marques four my revue
Miss steaks eye kin knot sea

Eye strike a quay and type a word
And weight four it two say
Weather eye am wrong oar rite
It shows me strait a weigh

As soon as a mist ache is maid
It nose bee fore two long
And eye can put the error rite
Its rare lea ever wrong

Eye have run this poem threw it
Eye am shore your pleased two no
Its letter perfect all the weigh
My chequer tolled me sew.

Figure 1. Test file for **spellchk**

`FILE *yyin;`	Is the file from which **yylex** reads its input. It must be opened before calling **yylex** to read any file. If it is not **stdin**, then it should be closed when **yylex** is done with the file.
`char *yytext;`	Whenever **yylex** returns from scanning, this variable points into its buffer at the word just recognized.
`int yy_flex_debug;`	This is a debug flag which puts **yylex** into verbose mode. You probably don't need it.
`int yylex (void);`	Scans the file **yyin** and for each word found, returns a non-zero integer code, leaving **yytext** to point at the word. It returns 0 at end of file.
`int yylineno;`	is used by **yylex** to keep track of the current line number, which is useful to you in error reporting. It needs to be reset to 1 before beginning a scan.
`void yycleanup (void);`	Cleans up the buffers allocated by **yylex** and releases their storage. This is called only when **yylex** is no longer needed.

Figure 2. Interface to the function **yylex()**

3. Survey of the code

Study the existing code, which is a partial implementation of your program. There are several modules and other files, most with a separate **.h** and **.c** file.

(α) The default dictionary really ought to be **/usr/share/dict/words**, and would actually be in a real implementation, however it is very large on Linux, so a smaller dictionary was copied from a Solaris machine. Using the smaller dictionary will allow you to test your program without needing to use a very large amount of memory, expecially when debugging. See the output of **wc**(1):

```
  25143   25143  206663 cmps012b-wm/usr/dict/words
 479623  479623 4950996 /usr/share/dict/words
```

(β) **Makefile** contains the usual building information. **Makefile.deps** is a generated file which lists the dependencies. It must be regenerated any time there is a change to any **#include** statement in your program.

(γ) The program **pspell.perl** is a reference implementation, but does not contain debug switches.

(δ) The module **debugf** contains some useful utility functions which are generally useful for debugging. See **debugf.h** for a description of each function.

(ε) The module **hashset** will require the most work. A stub has been provided, along with a hashing function **strhash**. You will need to add functions that print out the required debugging and also replace the calls to **STUBPRINTF** or **DEBUGF** with working code.

(ζ) The file **scanner.l** contains a scanner which reads words from the files to be tested. You need not understand exactly how this works, just how to call it to extract words. The file **yyextern.h** has definitions of the files in the program generated by the scanner. See Figure 2 for a description of the variables.

(η) The main program, **spellchk**, will scan the options, load the dictionaries, and then do the spell checking.

4. Implementation — Loading the dictionaries

Before implementing the hash sets, the dictionaries must be loaded into memory.

(α) There are two dictionaries to be loaded. Look for the stub which prints the message "Load dictionaries". Replace this with a loop that loads the dictionaries into the hash set.

(β) The variables **default_dictionary** and **user_dictionary** contain, respectively, the names of the default and user dictionaries. A null value indicates that the dictionary should not be loaded. Print an error message and stop if neither dictionary is present.

(γ) Create a **hashset** and iterate over each dictionary. Read each line using **fgets**(3c) and chomp off the trailing newline character. Then call **put_hashset** with each word.

(δ) Test your program. Use **valgrind** to find problems with uninitialized pointers and other bad memory references. Ignore any complaints it makes about memory leak. Fix any problems with segmentation faults or bus errors or other problems reported by **valgrind** and **gdb**.

5. Implementation — The hash set

Allocation and freeing of the hash set has been implement, but not insertion and searching.

(α) A function **strhash** has been provided which takes a string and, using Horner's method, iterates over the string to compute $\sum_{i=0}^{n-1} c_i \times 31^{n-i-1}$ where c is the integer coded value for each character in the string. Since overflow happens with longer words, we avoid negative numbers by using the defined data type **hashcode_t**, which is defined to be an unsigned 32-bit integer. You must declare any variable that is the result of the function **strhash** of this type, and not **int**.

(β) To search the hash set, compute the hash code first, then take it remainder the length of the array :

```
starting_index = strhash (word) % hashset->length;
```

(γ) Starting from the index thus given, search forward using linear probing of successive elements of the array (use **strcmp**(3c)) until you find a null pointer or an equality. The array is to be considered circular, so after looking at the last element of an array, wrap around to element 0 of the array. Make sure you don't have an infinite loop.

(δ) To put a new word into the hash set, first hash it, and then search forward exactly as you did for **has_hashset**. If the word is found and already there, ***do not*** insert it ; just return. If you find a null pointer before finding the word, store the address of the word in the hash set.

(ε) At any time if the **load * 4 > length** (the number of words in the hash set is more than a quarter the length of the array), perform array doubling. A hash modulus works better when it is not a power of two, but $2^n \pm 1$ works fine, which is why the initial size was 15. To double the array, the new size should be $2n+1$ if the previous size was n. This means that the sequence of sizes is 15, 31, 63, 127, 255, 511, 1023, 2047, 4095, 8191, 16383, 32767, 65535, etc.

(ζ) To perform doubling, allocate new array, then iterate over the old array, recompute the hash number modulo the new size, and insert it into the new array. After this is done **free**(3c) the old array. Note that initial insertion does a **strdup**(3c), but here, only the pointer is copied.

(η) Implement the debugging requirements. For the **-x** option given once, print out a table like the following, with numbers actually gleaned from the hash table.

```
 25143 words in the hash set
176001 length of the hash array
 25000 clusters of size  1
   100 clusters of size  2
    40 clusters of size  3
     3 clusters of size  7
```

A cluster is any run of adjacent non-null pointers. Print out a cluster message for each different number of cluster sizes. This verifies that your hash table is reasonably implemented.

(θ) If the **-x** option is specified more than once, follow this clustering information with an actual dump of the hash set, printing out each entry with subscript, hash code, and string, as with the line

```
array[        20] =      2489332 "foobar"
```

Print out only those entries that are not null. Use the format string

```
array[%10d] = %12u \"%s\"\n
```

6. Implementation — The final stages

At this point the program is almost complete. A few more things need to be done.

(α) The scanner **scanner.l** is written in the **flex** language and extracts words from a file that has been opened for the **FILE*** variable **yyin**. It is used to generate the C module **scanner.c**, the contents of which you do not need to read. Just treat it like any other library module.

(β) Implement the function **spellcheck** so that for each word that is found, the word is looked up in the dictionary. If it is not found, convert the word to lower case (**tolower**(3c)), and then look it up again. This is done so that capitalized words, such as at the beginning of a sentence, will be found. Proper names in the dictionaries are given in upper case, so will not be found if spelled in lower case.

(γ) Make sure the exit status of the program is correct as defined in the man page at the beginning of this document. Run **gdb** and **valgrind** to verify that you have no memory problems.

(δ) If you have time, eliminate memory leak by freeing up all of storage. Check the errors file generated by **valgrind**.

(ε) Use **checksource** and **lint** to verify good coding style. Read **Coding-style/**.

7. What to submit

README, **Makefile**, **debugf.h**, **hashset.h**, **strhash.h**, **yyextern.h**, **debugf.c**, **hashset.c**, **spellchk.c**, **strhash.c**, **scanner.l**. Do not submit the file **scanner.c**, which will be created when **gmake** is run. Do not submit any files that are built by **gmake**. Verify that the **submit** target in the **Makefile** is in fact correct and really does submit all files needed to build the target and does not submit any files that need to be generated. If you are doing pair programming, carefully follow the instructions in **/afs/cats.ucsc.edu/courses/cmps012b-wm/Syllabus/pair-programming/**.