$Id: asg1-stringtab.mm,v 1.36 2011-09-22 17:58:21-07 - - $
/afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments

1. Overview

Write a main program for the language **oc** that you will be compiling this quarter. Also, write a string table ADT for it, and make it preprocess the program using the C preprocessor, **/usr/bin/cpp** The main program will be called from Unix according the usage given below under the synopsis. This means that your compiler will read in a single **oc** program, possibly with some options, as described below.

The name of the compiler is **oc** and the file extension for programs written in this language will be **.oc** as well. Option letters are given with the usual Unix syntax. All debugging output should be printed to the standard error, ***not*** the standard output. Use the macros **DEBUGF** and **DEBUGSTMT** to generate debug output. (See the example **expr-smc**, module **auxlib**).

SYNOPSIS

oc [**-ly**] [**-@** *flag* . . .] [**-D** *string*] *program***.oc**

OPTIONS

-@*flags* Call **set_debugflags**, and use **DEBUGF** and **DEBUGSTMT** for debugging. The details of the flags are at the implementor's discretion, and are not documented here.

-D*string* Pass this option and its argument to **cpp**. This is mostly useful as **-D__OCLIB_OH__** to suppress inclusion of the code from **oclib.oh** when testing a program.

-l Debug **yylex()** with **yy_flex_debug = 1**

-y Debug **yyparse()** with **yydebug = 1**

Besides the debug options, your compiler will always produce output files for each assignment. Whenever your compiler is run for any particular project, it must produce output files for the current project and for all previous projects. Note that since *program* is in italics, it indicates that you use the name specified in **argv**. Your compiler will work on only one program per process, but it will be run multiple times by the grader and each run must produce a different set of output files.

asg 1	write the string table to	*program***.str**
asg 2	write each scanned token to	*program***.tok**
asg 3	write the abstract syntax tree to	*program***.ast**
asg 4	write the symbol table to	*program***.sym**
asg 5	write the intermediate language to	*program***.oil**

The first project will produce only the **.str** file. The second project will produce both the **.str** and **.tok** files. Each subsequent project will produce the files of all previous projects and also the one for the current project. Do not open output files for projects later than the one you are currently working on.

The main program will analyze the **argv** array as appropriate and set up the various option flags. *program***.str**, depending on the name of the program source file. Created files are always in the current directory, regardless of where the input files are found. Use **getopt**(3) to analyze the options and arguments.

The suffix is always added to the basename of the argument filename. See **basename**(1). The basename is the argument with all directory names removed and with the suffix (if any) removed. The suffix is everything from the final period onward. Be careful to not to strip off periods in the directory part of the name. An error is produced if the input filename suffix is not **.oc**, but compilation continues anyway. If there is no suffix in the basename, the output filename suffix is just appended. ***Note:*** This means that your program must accept source files from a directory that you do not own and for which you have no write permission, yet produce output files in the ***current*** directory.

2. Organization

The main program will call a test harness for the string table ADT. The test harness will work as follows: after filtering the input through the C preprocessor, read a line using **fgets**(3), and tokenize it using **strtok_r**(3), with the string "**\ \t\n**", i.e., spaces, tabls, and newline characters, and insert it into the string table. After that, the main program will call the string table ADT opration to dump the string table into its trace file. See the example in the subdirectory **cppstrtok** for an illustration of how to call the C preprocessor. Your program will not read the raw file, only the output of **cpp**.

Do not confuse the program **cpp**, which is the C preprocessor with the suffix **.cpp**, commonly used to indicate a C++ program, compiled via the **g++** compiler.

The purpose of the string table is to keep tracks of strings in a unique manner. For example, if the string "**abc**" is entered multiple times, it appears only once in the table. This means that instead of using **strcmp**(3) to determine if two entries in the hash table are the same, one can simply compare the pointers.

This assignment does ***not*** involve writing a scanner. Your dummy scanner, part of the main program, will just use **fgets**(3) to read in a line from the program file, and use **strtok_r**(3) to tokenize it, and then enter the token into the hash table.

3. The String Table ADT

The string table will operate as a hash table and have the interface in a file called **stringtable.h** and the implementation in **stringtable.c** As you develop your program, other functions may be needed. Use appropriate abstract data type (ADT) style with information hiding where possible and appropriate. For this assignment, that means that no structures may be declared in a header file, only handles.

Following is the interface specification. You may alter it in minor ways as needed if you find the interface to be somewhat inconvenient. You may not, however, violate ADT information hiding. Remember that you need file guards.

typedef char *cstring;
: (Obvious?)

typedef uint32_t hashcode_t;
: Type type **uint32_t** is defined in **<inttypes.h>**

typedef struct stringtable *stringtable_ref;
: A handle pointing at the entire hash table. Returned by the constructor and freed by the destructor. The structure is in the implementation file and consists of a pointer to the array of hash headers and a dimension. Also, a pointer to an internal static string identifier so that an assertion can be used to check the validity of the node. Also, the number of elements in the table is cached in this node in order to determine when the table needs doubling. Other fields if needed.

typedef struct stringnode *stringnode_ref;
: A handle pointing at an individual string node. The node itself contains a pointer to a string on the heap, along with a collision resolution pointer and a cached hash number. Also a pointer to an internal static string identifier.

stringtable_ref new_stringtable (void);
: Creates a new string table and returns its handle. Uses as its initial capacity some arbitrary odd number, such as 31.

void delete_stringtable (stringtable_ref);
: Dismantles the entire hash table and use **free**(3) to delete all internal nodes. Implemention of the function is optional.

void debugdump_stringtable (stringtable_ref, FILE*);
: Dumps out the hash table in debug format:

```
24    348883689   "hello"
     4294967295   "there"
92    338729983   "67"
```

In other words, print the hash header number in `%8d` format followed by spaces, then the hash number (`%12u`), and then the strings in `%s` format inside quotes in a column. In the above example, the two strings in bucket 24 have collided. It will always be true that the second number modulo the size of the hash table will be equal to the first number.

`stringnode_ref intern_stringtable (stringtable_ref, cstring);`
Interns the argument string into the string table and returns a pointer to the internal node. The client is honor-bound not to preserve this pointer beyond the life of the hash table. An assertion is used in other functions to verify membership. There is no difference between insert and lookup. If the key is not found, it is inserted then returned. If it is found, the existing node reference is returned.

The argument string is copied onto the heap with **`strdup`**(3), since the second argument is a loan argument, not a transfer of ownership argument. You may of course use **`malloc`**(3) and **`strcpy`**(3) instead. The hash number is also cached in the node to avoid having to recompute it.

`cstring peek_stringtable (stringnode_ref);`
Returns a pointer to a character string, such as may be used for printing.

`hashcode_t hashcode_stringtable (stringnode_ref);`
Returns the cached hash number associated with this string node. Does not recompute the number.

4. The hashing function

The old 1986 ***Dragonbook*** (Aho, Sethi, Ullman), page 436, lists P.J. Weinberger's hashing function **`hashpjw`** as one the authors recommend. Another hashing function that may be used is at **`http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html#hashCode()`**. In either case, make sure that the result of your hashing function is a **`hashcode_t`** value. Also see **`misc-code/strhash.h`** (Figure 1) and **`misc-code/strhash.c`**(Figure 2). An example of a trivial compiler is in **`/afs/cats.ucsc.edu/courses/cmps104a-wm/Examples/E08.expr-smc/`**

5. Collision resolution

Obviously, your hash table implementation must provide collision resolution. There is no delete operation, only an intern operation, which is an insert if the key does not already exist, but a lookup if it does. You may use whatever collision resolution method you prefer. Note that in either case, array doubling is ***required***.

Begin your table with the hash array dimension as 31. Then every time you double the array size, increase it to the next number which is of the form $2^k - 1$. That is, use the sizes 31, 63, 127, 255, 511, 1023, etc. Using a modulus wich is near but not a power of two scrambles the hash number a little better by using all of the bits of the number, rather than just the low order bits.

Collision resolution by separate chaining means that the has table contains a pointer to an array of collision resolution chains, each of which points at a linked list. If the hash table's loading factor exceeds 0.75, the array of hash headers is doubled. The loading factor is the number of nodes in the table divided by the number of headers. To double the header array, allocate a new copy, and then copy each node from the old header array to the new header array. Then free the old header array. Note that the strings are not rehashed. Only the modulus is recomputed. When the array is doubled, make sure the result is an odd number (add 1). While this operation takes $O(n)$ time when it is done, it is amortized to take $O(1)$ time.

Open addressing with linear probing means that the hash array is a direct pointer to a hash node, and no linked lists are used. Instead, if a key is not found at a given position, one scans down the array, wrapping around at the end, until either the key is found or a null is found. In this case, it is critical that the table be lightly loaded, since large clusters will produce poor performance. The load

factor here should be definitely always less than 0.5, although if the same amount of space is to be wasted on null pointers, then a maximum load factor would be 0.75/1.75.

This discussion is done here for the purposes of a review and is necessarily incomplete. You are assumed to have studied hash tables in your Data Structures and Algorithms courses, which are prerequisites for this course. Note that the Java API uses the term **`HashSet`** rather than hash table.

```
 1
 2    //
 3    // NAME
 4    //    strhash - return an unsigned 32-bit hash code for a string
 5    //
 6    // SYNOPSIS
 7    //    hashcode_t strhash (char *string);
 8    //
 9    // DESCRIPTION
10    //    Uses Horner's method to compute the hash code of a string
11    //    as is done by java.lang.String.hashCode:
12    //    .  s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
13    //    Using strength reduction, the multiplication is replaced by
14    //    a shift.  However, instead of returning a signed number,
15    //    this function returns an unsigned number.
16    //
17    // REFERENCE
18    //    http://java.sun.com/j2se/1.4.1/docs/api/java/lang/
19    //    String.html#hashCode()
20    //
21    //
22
23    #ifndef __STRHASH_H__
24    #define __STRHASH_H__
25
26    #include <inttypes.h>
27
28    #include "auxlib.h"
29
30    typedef uint32_t hashcode_t;
31
32    hashcode_t strhash (char *string);
33
34    // LINTED(static unused)
35    RCSC(STRHASH_H,"$Id: strhash.h,v 1.1 2011-08-31 19:29:11-07 - - $")
36
37    #endif
38
```

Figure 1. `misc-code/strhash.h`

6. Filenames

The following project organization rules apply to everything you submit in this course, in order to ensure consistency across all projects, and to make it easier for the grader to figure out what your compiler is doing (or not doing). You may use any development environment you wish. However, the production environment is that available under **`unix.ic`** As regards grading, whether or not your program works on the development environment is not relevant. The grader will use only **`unix.ic`** to

```
 1
 2    #include <assert.h>
 3
 4    #include "strhash.h"
 5
 6    hashcode_t strhash (char *string) {
 7       hashcode_t hashcode = 0;
 8       assert (string != NULL);
 9       for (;;) {
10          hashcode_t byte = (unsigned char) *string++;
11          if (byte == '\0') break;
12          hashcode = (hashcode << 5) - hashcode + byte;
13       };
14       return hashcode;
15    }
16
17    // LINTED(static unused)
18    RCSH(STRHASH_C,"$Id: strhash.c,v 1.1 2011-08-31 19:29:11-07 - - $")
19
```

Figure 2. **misc-code/strhash.c**

test your programs. Use the Solaris submit command to submit your work.

Any special notes or comments you want to make that the grader should read first must be in a file called **README**. Spell it in upper case. The minimum **README** should contain your personal name and username, and that of your team partner, if any.

Use of **flex** for the scanner and **bison** for the parser is required.

Compile your hand-coded programs with

gcc -g -O0 -Wall -Wextra -std=gnu99

and make sure that the programs are fixed so that no warning messages are generated. Compile the programs generated by **flex** and **bison** using whatever options will cause a silent compilation. Also see **Examples/E08.expr-sm/Makefile** If you prefer using **gcc** instead, that is OK, but remember that you must then use **gdb** as the debugger. ***Run* lint *frequently!***

You must submit a **Makefile** which will build the executable image from submitted source code. If the **Makefile** does not work or if there are any errors in your source code, the result of which is a compilation failure, you lose all of the points for program testing.

The executable image for the compiler you are writing must be called "**oc**". Use appropriate source file suffixes :

- **.c** for C source code.
- **.l** for **flex** grammars.
- **.y** for **bison** input grammars
- **.h** for all header files, both C and C++.

You may use C++ if you are an expert programmer in that language, in which case your implementation files must end in either **.cpp** or **.cc**. Note that **flex** and **bison** generate C code, not C++ code, so your C++ program will have to link with C code. If you want to make these generators generate C++ code, RTFM. **Warning :** Only the C versions will be covered in lectures, so you are on your own with the C++ versions.

Your compiler must accept input filenames with arbitrary directory specifications and filetype suffixes. Files created by your compiler must always be created in the current directory and have the suffix replaced as appropriate. Example, an input file of **/foo/bar/baz.oc**should generated an output file of **baz.str**, **baz.tok**, etc. The input suffix may be anything at all, including nothing.

7. Makefile

You must submit a `Makefile` with the following targets :

- `all:` Build the executable image, all necessary object files, and any required intermediate files. This must be the first target in the Makefile, so that the Unix command `gmake` means `gmake all`
- `clean:` Delete object files and generated intermediate files such as are produced by flex and bison. Do not delete the executable image.
- `spotless:` Depends on `clean` and deletes the executable image as well.
- `lint:` Runs `lint`(1) on all your source files. See the sample `Makefile` for suitable options. "Thou shalt run `lint`(1) frequently and study its pronouncements with care, for verily its perception and judgement oft exceed thine." ["The Ten Commandments for C Programmers" (annotated ed.), Henry Spencer. — `http://www.lysator.liu.se/c/ten-commandments.html`]
- `ci:` Checks in all source files (but not generated files) into the `RCS` subdirectory. You may use `SCCS`, `CVS`, or `SVN` if you prefer.
- `deps:` Recreates the dependencies.

8. What to submit

`README`, `Makefile`, `stringtable.h`, `stringtable.c`, `main.c`. You may also need to submit things like `boolean.h`, `auxlib.[hc]`, or other utilities that you need.

Warning: After you submit, you ***must*** verify that the submit has worked. Make a new empty directory in your personal file space, copy all files from the submit directory back into yours and perform a build. Failing to submit a working build will cost you 50% of the points for an assignment. It is not a "simple" mistake if you forget. You just ***don't*** forget such a thing if you want to pass the course.

Also, ***use RCS***! Or something similar to maintain backup copies of your source code. You may wish to periodically archive your project into a `tar.gz` in order to keep copies. If you are working with a partner, keep a backup copy in a place your partner has no access to. If you partner accidentally deletes all source code on the due date, you get a zero as well.