$Id: asg1-stringset.mm,v 1.14 2013-10-03 18:14:18-07 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments
URL: http://www2.ucsc.edu/courses/cmps104a-wm/:/Assignments/

1. Overview

Write a main program for the language **oc** that you will be compiling this quarter. Also, include a string set 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 set 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**, or if there is no suffix, in which case the compilation aborts with a message. ***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 set 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 set. After that, the main program will call the string set ADT opration to dump the string set 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 set 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 Set ADT

The string set will operate as a hash table and have the interface in a file called **stringset.h** and the implementation in **stringset.cc**. As you develop your program, other functions may be needed.

Following is the interface specification. You may alter it in minor ways as needed if you find the interface to be somewhat inconvenient.

const string* intern_stringset (const char*);

Insert a new string into the hash set and return a pointer to the string just inserted. If it is already there, nothing is inserted, and the previously-inserted string is returned.

void dump_stringset (FILE*);

Dumps out the string set in debug format, which might look like the following:

```
stringset[   3]:    2586491021746226264 0x9903d0->"teststring"
                   12271277041006505511 0x990240->"main.o"
stringset[  13]:   18201842504327843073 0x990150->"Makefile"
load_factor = 0.522
bucket_count = 23
max_bucket_size = 2
```

In other words, print the hash header number followed by spaces, then the hash number and then the address of the string followed by the string itself. In the above example, the two strings in bucket 3 have collided.

4. 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 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

```
g++ -g -O0 -Wall -Wextra -std=gnu++0x
```

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`. Run `valgrind` frequently to check for uninitialized variables.

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 :

- `.cc` for C++ source code.
- `.l` for `flex` grammars.
- `.y` for `bison` input grammars
- `.h` for all header files.

5. 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.
- `ci`: Checks in all source files (but not generated files) into the `RCS` subdirectory. Or you may use `SCCS`, `CVS`, `SVN`, `Git`, or some other archival system that you find convenient.
- `deps`: Recreates the dependencies.

6. What to submit

`README`, `Makefile`, and all C++ header and implementation files. Ensure that all files needed to build the project are submitted. In later projects, ***do not*** submit files generated by `flex` and `bison`. When the grader types the command `make` in the submit directory, the executable binary `oc` should be built. No error messages or warnings should be printed.

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.