Part 2-Turnin proj1-part2 due 7:30 PM Wednesday, February 17.

All code must be maintained with RCS as you develop it. Only the RCS database files are to be submitted. See the Project1-Part1 assignment for instructions. If these instructions are not followed, the project will not count!

NO cores, executables, or object files!!
Now that you are used to turnin, we will begin to enforce that rule that if your submission contains “core” files, executable files, or object files, either inside RCS database files or as plain files, 10 points will be deducted from the project score. One reason is that we compile and examine only the sources to grade the projects, but the main reason is that these unnecessary files use up space allocated for the class account. When the class account space allocation runs out, work turnins of other students fails.

Normally, a core file is written to the current directory when your process “crashes.” “Crash” here means that the operating system detects (from a hardware action called an interrupt) that the process tried to execute an invalid instruction, or tried to access memory at a virtual address that was not mapped for the kind of access tried. Student programs usually “crash” when an invalid pointer values such as 0 (i.e., NULL) is dereferenced. The “core” file is a snapshot copy of process’s virtual memory contents at the time of the crash. It is useful for finding the cause of the crash when the crash conditions cannot be reproduced when re-running the program under the debugger. Since core files are large (several megabytes, typically) they use up disk space quota fast.

Tip: Add to the bottom of the file named .cshrc in your home directory the line\(^1\)

```
limit coredumpsiz e 0k
```

Now, when you log in again (or just “source” the file .cshrc, the system will not write core files after crashes of your processes.

Turn in your RCS files to proj1-part2 with the commands
```
turnin-csi402 -p proj1-part2 README RCS/*
turnin-csi402 -v -p proj1-part2
```

Check that you submitted ALL files needed to compile and link your program successfully.

Late submissions are welcome and will count according to the following formula: \( G = \max(0, OG \cdot (7 - ND)/7) \) where \( OG \) is the grade you would have gotten had the submission been on time, \( ND \) is the number of days past 7:30 PM Wednesday, and \( G \) is the grade that your project will get, including the late penalty.

The goals of Part 2 are:

- Program with some system library functions or facilities and their data types and structures, which requires that your read and understand their specifications.

- More practice building multiple module C++ programs.

\(^1\)We set up the ECL so that the coredumpsiz e limit was by default set for all users, so beginning students didn’t have to worry about it.
• Learning about run time information passed to program in the (Unix) environment.

• Implement a simple lexical scanner controlled by a finite state transition and action table; use of C/C++ structure and array initializers and function pointers. For information on how to build tables into your program which is a use for C/C++ initializers, see Harbison and Steele, pages 95-100. An introduction to initializers appears in Deitel and Deitel section 4.4 (“Examples using arrays”).

• Implement string variable substitution in strings.

• Learn about a few of the elements that go into a command interpreter or shell or script interpreter. You can read more about shells in the Topics in C book and most other books on Unix systems programming.

Table driven scanning and variable substitution will count 50%, and the rest (specifications 1-10) counts for 50%.

**What your program should do (specifications)**

1. Develop, maintain under RCS, use and submit an executable shell script file (NOT a Makefile) which when executed, will build the executable specified below from sources in the current directory. The executable must be named part2

2. When part2 is run, it should print the line

   CSI402 Project 1 Part2 work by <your name>, February 1999.

   where of course <your name> is replaced by your real name. It should then run the main loop that begins by printing the prompt:

   >>>>

   (there is one blank after the >) It should handle user input lines just as the Part 1 skeleton program did. For simplicity, if a command that takes a certain number of arguments is given too many, ignore the excess.

3. Maintain the dictionary that is manipulated and tested by the commands insert, retrieve, delete, and dump, as specified in Part 1.

4. Add the command **exit** that makes the program exit.

5. Add the command **tokens** which prints the tokens that follow on its input line each on its own line with the token number. Follow the specifics of the example below:

   >>>> tokens T0ken1 SecondToken 3rdToken
   token[0] : tokens
   token[1] : T0ken1
   token[3] : 3rdToken
(Use cout << "token[" << n << "]": " << pch << endl;
where pch is of type char *. What is recommended for further steps is to grow an array
of char * to hold pointers to each token. Note that the command itself is the first token.

6. Add the command ls that prints the name of each file in the current directory, one file
name per line, left justified.

Read the man page named directory and use the Unix system library functions opendir,
readdir and closedir to implement ls. (Use command: man directory)

7. Add a “stopwatch feature.” A stopwatch accumulates a time quantity when it is on. It
holds the time quantity constant when it is off. Each stopwatch command reports the
time quantity and performs the action or reports the errors described below.

However, on a multiprogrammed computer system there are three notions of “time.” A
multiprogrammed computer system shares the processor (CPU) among several different
PROCESSES, switching between them so that each user of interactive software gets the
illusion that the computer is working for him or her exclusively. So, first, there is the
familiar “wall clock time” that records ordinary physical time. This is the time the
user feels. Second, there the amount of time that the CPU spends on executing
the instructions of the particular user’s run of his or her program. (The run of one program
is called a PROCESS.) Third is the amount of time used by OPERATING SYSTEM
CPU instructions to provide services called for by the particular process. (Waiting for
I/O devices is not counted here.)

Hence, your stopwatch will accumulate and report three time quantities. Each quantity
will be reported in units of “clock ticks.” Clock ticks are used in the time reports
provided by the standard POSIX function times. Read about times in the GNU C
library documentation you can get from emacs as follows:

1. In emacs, type C-h i
   This will show a top level menu page of the emacs “info” hypertext browser.

2. Find the item
   * Glibc: (libc.info). Documentation for the GNU Standard C runtime library.
   in this page. Position the cursor on this line and press enter.

3. Look for an item Date and Time on the libc page. Select it as you did for libc.

4. Now look under the Processor Time item.

You can also use search features of the emacs info system.

You can also find documentation for times from Section 2 of the on-line manual pages.
Use the command “man -s 2 times”

Each stopwatch command should first print any message specified below under the spec-
ified conditions. (When the stopwatch is used “correctly,” no message is printed.)

---

2The clock tick is the interval between which a timer hardware device interrupts the application computing
so the operating system gets a chance to switch which process is using the CPU.
Second, after possibly printing a message, the stopwatch facility should print a report in the form:

```
real ticks=nnnnn, user ticks=nnnnn, system ticks=nnnnn
```

where each sequence of n’s represents a decimal integer numeral.

- **stopwatch** (no arguments) Just report the time quantities.
- **stopwatch -on** Resume accumulating the time quantities, i.e., make the stopwatch go on. If the stopwatch is already on, print the message *Already on*.
- **stopwatch -off** Stop accumulating the times, i.e., make the stopwatch go off. If the stopwatch is already off, print the message *Already off*.
- **stopwatch -zero** If the stopwatch is off, reset all the time quantities to zero. Report the time quantities as 0 each. If the stopwatch is on, print the message *Must be off to reset stopwatch*. Then, print the current quantities and do not stop or reset.

The stopwatch is initially off and all three quantities are initially zero.

8. Add the command `printenv` that prints the environment just like the Unix shell command `printenv` does. Sometimes, some shells will make your program get an environment that contains one or more null (zero length) strings. Such null strings must be skipped by `printenv` and by all the commands below that refer to the environment.

9. Make your `printenv` command check for an argument `-v`

   When so, it prints the environment name and value strings on separate lines after labels that number them. Each name/value pair is separated from the next by a blank line. For example, if (your or the Unix) plain `printenv` prints

   ```
   PWD=/home/classes/csi402
   USER=sdc
   ```

   your `printenv -v` command should print:

   ```
   Variable 1: PWD
   Value 1: /home/classes/csi402
   ```

   ```
   Variable 2: USER
   Value 2: sdc
   ```

   These examples specify the spacing that must be followed.

   Very important: DO NOT MODIFY the original environment strings.

10. Add the command `import` that copies the environment into the same dictionary maintained by `insert, retrieve,` etc. For each (non-null) environment string, the key is the substring that precedes the first `=` (equal sign) and the value is the substring after the first equal sign. Watch out that environment values can contain one or more `=`’s (Try `printenv TERMCAP` in a real shell on `eve`’s.). Check and skip null strings that sometimes appear in the environment.
Be sure to learn the difference between a “null pointer value” and a “pointer to a null string,” if you don’t know it yet!

Table Driven Scanning

The next 4 items are to be implemented using a table driven finite state scanner that will be covered in class. To make this an independent part of the project and keep the state table design small, the table driven scanner should operate on individual quoted tokens one by one.

I hope Beck’s text comes in time for this; if not, I'll supply additional materials. The relevant section in Beck is 5.1.2 on Lexical Analysis.

11. Add the commands set scandebug and set -scandebug. For all commands subsequent to a set scandebug (without an intervening set -scandebug), the scanner will report its progression from state to state as it processes characters. The report must be of the form:

    Init state=START
    Char='x' Class=Ordinary
    Next state=INTOK

(etc...). Here, Init state=, Class=, Char=' , etc. must be printed literally, but the state and character class names here are just examples. The names of the character classes and the states are those that you use in your scanner.

Naturally, set -scandebug turns off scanner reports until the next set scandebug.

(You can add other set somethingdebug commands to enable other debugging output of your choice if you wish.)

12. (Quoted tokens) A shortcoming of whitespace (or any other!) token delimiters is that a token cannot contain any whitespace (or other delimiter) characters. Also, delimiting by one or more whitespaces prevents the empty string from being a token. Indeed, C/C++ uses delimiting by whitespaces; this is one reason C/C++ string constants are written between double quotes.

Make your program accept as a single token all the characters (other than "" for now) that lie between matching pairs of double quotes. The tokens command will be used to test this feature. For example:

    >>>> tokens First       "Around Me ,", I like " <spaces-> " dot.
token[0]: tokens
token[1]: First
token[2]: Around Me ,
token[3]: I
token[4]: like

*Fortran IGNORES whitespaces!
token[5]:  <-spaces->  
token[6]:  dot.  
>>>> tokens Empty string now-> "" (invisible)  
tokens[0]:  tokens  
tokens[1]:  Empty  
tokens[2]:  string  
tokens[3]:  now->  
tokens[4]:  
tokens[5]:  (invisible)  
>>>>  

Observe that inside the quoted tokens, all originally entered spaces are retained.  
For simplicity, all tokens will be separated by whitespace characters on input. With input like  

>>>> tokens  NoSpace"Quote here"  

program behavior is unspecified, but should be “reasonable” (not crash, not loop infinitely,  
etc.)  

13. Make sure  
>>>> "tokens"  bla  blah  
is equivalent to >>>> tokens  bla  blah  

14. Detect and report the error when an input line ends before a quoted token is complete.  
Use the specific message below (or the grading script will deduct points). Example:  

>>>> tokens  Try starting "a string here  
Unmatched "  

15. Now it is impossible for a quoted token to contain a quote character! So, make backslash  
(\) be the “escape character” within a quoted token. For example,  

>>>> tokens "See  ->"<- here"  
token[0]:  tokens  
token[1]:  See  ->"<- here  
>>>>  

But now, a backslash cannot be put in a token. Therefore, make backslash be a one  
character escape for all characters. Specifically, if a backslash appears inside a quoted  
token, the backslash will be removed and the following character included without special  
properties into the string produced for the token. For example,
16. Let the “identifier characters” be non-escaped letters, digits and underscore character. An “identifier” is a non-empty string of identifier characters.

Within quoted tokens (only), make the scanner recognize the maximal string of identifier characters that follow immediately after a non-escaped dollar sign $; do it for each non-escaped $. Thus every identifier following every non-escaped $ will be found. An non-escaped $ that is not followed immediately by an identifier is an error for which the Illegal variable message shown below should be printed.

For each identifier found this way, look it up in the dictionary. If any of these identifiers is not in the dictionary, print the Undefined variable message as shown below.

The string composed of the $ and the (maximal) identifier that follows it is called a “variable reference.”

If all the variables are legal and defined (i.e., in the dictionary) report for that token the string obtained from the input by replacing every variable reference (substring) with its value (string).

>>> set -scandebug
>>> insert user FamousName
>>> insert Var2 Value$Dollar
>>> insert Dollar you_wont_see_this
>>> tokens "Certain$user\'s spend $Var2" unquoted: $Dollar
 token[0]: tokens
 token[1]: CertainFamousNames spend Value$Dollar
 unquoted:
 token[2]: unquoted:
 token[3]: $Dollar
>>> delete user
>>> tokens "$user"
 Undefined variable $user
>>> tokens "A null string identifier is bad: $$Dollar"
 Illegal variable
>>> tokens "Another Bad $10ne (digit 1 here)"
 Illegal variable
>>> retrieve Dollar
 Key Dollar has value you_wont_see_this
Information about shells

This project introduces in simple form a few of the elements that go into a command interpreter or shell. A shell is a program that implements the interface between the user and the computer software system—It accepts commands that basically fall into two categories:

1. Commands to control or report information about the system. This includes the shell itself, files that are stored on disks, etc., programs that can be run, and information that will be available to programs when they are run.

2. Commands to start programs to run.

Nowadays, many people use operating systems that come with a *graphical user interface* shell. Popular examples are Windows 95/NT, the Macintosh operating systems, and Common Desktop Environment (CDE) that some of you used in the the prerequisite courses. The action of a user to “open an application” is really a shell command to start a program to run⁴.

For this course, you will learn the fundamentals with shells that are text based. In UNIX, command interpreters are NOT part of the operating system. Functions of shells include:

1. Reading and scanning of commands and their arguments that the user types.

2. Access and manipulation of the *argument list* and the *environment* which the operating system passes to programs when they begin to run, so that program operation can depend on data that is defined long after the program is written. These lists are implemented by of *ragged array* data structures. Here is an example of traditional Unix system programming where your program operates directly on data structures created by the system.

3. Searching for files to “execute” in directories listed in a *path*. This phase will use the more modern POSIX standard system library interface.

4. Substitution of environment variable values in shell commands. Shells also maintain separate dictionary of “shell variables” that can be substited in commands but are not passed to programs in the environmenet.

More Background

See *Topics in C Programming* pages 70, 71 and 72.

The second argument “*argv*” to *main* (see 6.14) is a pointer to a ragged array (see pp. 70-72). The third argument “*env*” is another pointer to a ragged array. See also the man pages for *printenv(1)*, *getenv(3)* and *environ(5)*. The (5) after “*environ*” means that it is in

⁴The Windows and Macintosh systems maintain associations between “types of files,” coded by the filename extension, and a particular program to run when that file or “document” is “opened.” Thus, end users need not learn about the concept of a “computer program” at all!
section 5 of the Unix manual. Since there is another man page \texttt{environ(4)}, you will have to use the command \texttt{man -s 5 environ} to view man page \texttt{environ(5)}. The command \enquote{setenv} belongs to the shell, see \texttt{csh(1)}. The man page for \texttt{environ(5)} seems to be missing from eve; I've requested \texttt{consult} to find it. (Kelly and Polh readers see Chapter 6, sections 6.13, 6.14 and 6.15 especially.)

Command line arguments and the environment are two ways ways fixed, prebuilt program operation can be made to depend on runtime information. Think of how silly it would be if, in order to print a file, you had to edit the source program of the print command (perhaps \texttt{1pr.c}) to insert the name of the file you want to print, recompile it (how would the compiler know what program to compile?) and finally run it?

What are some other ways a fixed program can get information at runtime? What about user interaction? When are command line arguments preferable to user interaction, and vise-versa? There are also more advanced ways: network or interprocess communication connections and other system calls. System calls invoke operating system services on behalf of a process. Actually reading and writing to files or to user interaction are special cases of system calls. System calls implement the features of the process’s \enquote{abstract} or \enquote{virtual} machine that are beyond the hardware’s instruction set architecture.

Two of the ways information is passed to a program when it begins to run are through command line arguments and through the environment. In this project you will learn how to make your programs access both the command line and the environment.

A \textit{process} is created when a the operating system loads an executable program. The loader (a component of the operating system) writes the machine language program into memory. It also makes available to the program a list of the arguments (which are C strings\footnote{Remember that a standard C string is null terminated.}) and a list of the environment strings.

**VERY IMPORTANT** for your grade and our sanity: Do not submit programs that really call \texttt{fork()} and \texttt{exec()}. Each legal environment string has the form

\[
<\text{NAME}> = <\text{VALUE}>
\]