EECS 2031 SOFTWARE TOOLS, SUMMER 2014
LAB 4

PRZEMYSLAW PAWLUK

1. Objective

Cellphones and other electronic devices must be off while you are in the lab.

Write an ANSI-C program called `lab3.c` that reads lines from standard input, parses them, and classifies the input according to the context set out above and the requirements specified below. Test that your program correctly implements the required functionality

Short Reference of useful Unix commands

First, open a command line window/unix prompt by starting an xterm.

1.1. Commands related to directories/folders.

- `ls` lists current directory
- `cd <name>` change current directory to named directory, `cd ..` to move up in the hierarchy, `cd ~` to go back to your home directory
- `pwd` prints location of current directory, i.e., the path where you currently are.
- `mkdir <name>` creates new subdirectory with the given name in current directory

Hint: you can use the tab key for autocompletion of file and directory names. The up/down cursor keys can be used to scroll through the history of commands.

1.2. Commands related to compiling files.

- `gcc -o <name> <name>.c` compiles the named source file `"name.c"` into the executable `"name"`. That executable can then be started by typing `"name"`.
- `cat <name>` prints the contents of a file to the terminal.

Note that you will need to re-compile your program before you can test changes.

2. What to do

First, create a few subdirectories:

- create a subdirectory called "lab4" in 2031
- change the current directory to the newly created directory
- if you print the current directory with `pwd`, the system should show you now something like `/cse/home/ZZZcseZZZ/2031/lab4` (with your CSE account name instead of `ZZZcseZZZ`)

Then create a simple ANSI-C program and compile it.

2.1. **Idea.** One common usage scenario for smartphones is to monitor the status of the user as it changes over time. It may contact various devices or services. Imagine that every time you're account balance changes the data is stored in a system and it can produce a file with history of changes. In this lab, we are going to implement a simple parser that deals with a set of data coming from such a service. Such data could be transferred in a simple text format.

Here we will deal with such input, consisting of individual change record, one per line. Each line contains the following information:

`timestamp userID newBallance`

There is a single Space character separating the 3 pieces of information. The fields are defined as follows:

- The timestamp is an integer with the number of seconds since 00:00, Jan 1, 1970 UTC, which conforms to the standard specification of time in Unix/Linux systems.
- The `userID` is a string, which conforms to the rules for the naming of C variables. It will be at most 31 characters long.
- The `newBallance` as a floating point number (`.2f`) representing new balance on the account.

2.2. **Requirements. Note:** We will work with the same file structure as in `lab3`.

(1) If the timestamp field is missing, is not an integer, or is zero, you must print Invalid time, followed by a newline character.
(2) If the timestamp of a record not larger than the previous timestamp, i.e., the current is in the past relative to the previous one or identical to it, you must print "Non-monotonic timestamps", followed by a newline character.
(3) If the first character of userid does not conform to the rules for C variable names or if the fields is missing, you must print Illegal userID, followed by a newline character.
(4) If the newBallance field is missing or is not numerical, you must print "Illegal balance", followed by a newline character.
(5) If the newBallance field is negative, you must print "Debit!!!", followed by a newline character
(6) If the balance change between two records that fit all above criteria exceeds 1000 $/day, you must print "Suspiciously large balance change", followed by a newline character. Added: You are welcome to ignore the userID for this test.
(7) If the information supplied in each line is otherwise fine, you must print OK, followed by a newline character.
(8) No other output must be produced.
(9) Additional text on the line after the last field should be silently ignored.
(10) If there are multiple problems with the line of input, you must print only the message for the first field that does not follow the specification. Processing should then continue with the next line. All rules need to be applied in the order specified here.

When EOF ($\hat{\text{D}}$) is encountered print the userID and final balance for each userID.

For the purpose of this lab, you do not need to worry about overflow. In other words, you can safely assume that timestamps are guaranteed to fit in 32 bit integers, userID's will not be longer than 31 characters, and floating point numbers will fit into a ANSI-C float variable. You can also safely assume that there is always a single space characters between the fields. Moreover, each line of input is guaranteed to be less than 99 characters long.

Assuming that the program is started with lab2, and given the following input, which is also provided for convenience as a file input.txt:

```
Testing 234 234.153
1235 Mega0123test -x-0.5
3600 godzilla 300
36000 godzilla 299
36001 godzilla 2000
36002 godzilla 0
36003 godzilla -10
1000 innocent 69
^D
```

your program should create the following output:

```
Invalid time
Illegal balance
OK
OK
Suspiciously large balance change
Suspiciously large balance change
Debit!!!
Nonmonotonic timestamps
godzilla -10
```

2.3. **Submission.** Submit your work using the submit command

```
submit 2031 lab4 lab4.c
```

2.4. **Hints.**

- Please do not use `scanf()` directly to parse the input. Instead read the input line by line and then use the string-parsing version of `scanf()`, `sscanf()`, to read the data.
- The scanf() family of functions returns the number of tokens/parts of the input that it was able to parse correctly. Using this fact will greatly simplify your code.
- It is usually easier to address each individual requirement to your code only after you have verified that the previous requirement is met by your program.
- Note that you need to test the rate of change ($/day) and not just the balance ($) itself.
- Note that you do not know the number of users in advance!