Project1a

Students will work in groups of two to complete the following tasks.

Goals

This project tests your ability to write very simple data-driven scripts and deliver that code to the lecturer.

Project1b asks you write some gawk.

But first, Project1a asks you to set up for Project1b.

Submission

Follow the instructions in How to submit using (say)

```
cd cs310/proj1a
make submit
```

Say "hello" to GRAPH

In this subject, you will implement the same program, four different ways (using data-driven languages, OO languages, functional languages, and logic languages).

The target program is "GRAPH", a tiny ascii GRAPH utility written by by Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. In inputs files that look like this:

```
label here's some stuff
bottom ticks 1 2 5 10
left ticks 1 2 10 20
range 1 1 10 20
height 10
width 30
1 2 *
2 4 *
3 6 *
4 8 *
```

And outputs stuff that looks like this:

```
20    -----------------------|
      |                      |
      |                      |
      |                      |
10    -       *              |
      |    *                 |
      |  *                   |
2     *--|------|------------|
     1  2      5            10
         here's some stuff
```

A sample implementation is supplied at http://unbox.org/wisp/var/timm/09/310/lib/proj1a/GRAPH.awk and understanding it will take up most of the time of project1.

Sample input files can be found at http://unbox.org/wisp/var/timm/09/310/lib/proj1a/eg. For example, the above input file is actually eg/1. To generate a graph, using GRAPH, run

```
gawk -f graph.awk eg/1
```

Graphs can contain multiple lines, so you can generate bar graphs as follows. This input:

```
label mean IQ
left ticks  0 10 20 30 40
height 20
width  30
3 1 *
3 2 *
3 3 *
3 4 *
3 5 *
3 6 *
3 7 *
7 1 +
7 2 +
7 3 +
7 4 +
7 5 +
7 6 +
7 7 +
7 8 +
7 9 +
7 10 +
7 11 +
7 12 +
7 13 +
range 0 0 10 20
tags  3 boys 7 girls
```

Generates this output (note the error message on line 1 saying that ***tags*** is not found

```
?? line 26: [tags  3 boys 7 girls  ]
20    -----------------------|
      |                      |
      |                      |
      |                      |
      |                      |
      |                      |
      |               +      |
      |               +      |
10    -               +      |
      |               +      |
      |               +      |
      |      *        +      |
      |      *        +      |
      |      *        +      |
      |      *        +      |
      |      *        +      |
      |      *        +      |
0     -----------------------|

              mean IQ
```

Annoyances (which you will fix)

Tags not supported

The current implementation does not add tags to bar graphs. The above output should have been:

```
20    -----------------------|
      |                      |
      |                      |
      |                      |
      |                      |
      |                      |
      |               +      |
      |              +      |
10    -              +      |
      |              +      |
      |              +      |
      |      *       +      |
      |      *       +      |
      |      *       +      |
      |      *       +      |
      |      *       +      |
      |      *       +      |
0     -------|--------|------|
             boys     girls
              mean IQ
```

(This desired output can be found at eg/6.want).

Tedious specifications

Another tedious feature of the current implementation is that the bar graphs have to be hard-coded using lines and lines of numbers. Ideally, we should be able to write bar graphs using the following shorthand:

```
label mean IQ
left ticks  0 10 20 30 40
height 20
width  30
range 0 0 10 20
tag 3 boys 7 *
tag 7 girls 13 +
```

Quirks (that you will accept)

Your OO/functional/logical versions of this this system should all produce the same output as the reference system. Except for the annoyances listed above, the reference implementation should be treated as ***the*** implementation.

This means that you must understand the reference system, warts and all, in order to reproduce it in other languages.

The Problem

The current version of GRAPH is broken, as revealed by the following test engine:

```
cd proj1
make score
?? line 12: [tags  2 things]
?? line 26: [tags  3 boys 7 girls  ]
?? line 6: [tags  3 apples 10 oranges 16 pears]
?? line 5: [tag 2 things 7 *]
?? line 6: [tag 3 boys 7 *]
?? line 7: [tag 7 girls 13 +]
?? line 3: [tag 3 apple 7 *]
?? line 4: [tag 10 oranges 24 +]
?? line 5: [tag 16 pears 16 $]
   6 FAILED
   4 PASSED
```

That is, the current system fails 6 of its current tests. You need to fix that. To do that, you'll need to:

1. Understand the GRAPH system
2. Set up the files
3. Understand the test engine
4. Understand what is causing the above failures
5. Fix the implementation.

Project1a is about 1,2,3. Project1b is about 4,5.

Setting up

1. Make sure you have a ***cs310*** directory.
2. Change to that directory.
3. Run this command:

   ```
   svn export http://unbox.org/wisp/var/timm/09/310/lib/proj1 proj1
   ```

   This should generate output that looks a little like this (some details may be different):

   ```
   A    proj1
   A    proj1/graph.awk
   A    proj1/eg
   A    proj1/eg/8.want
   A    proj1/eg/10
   A    proj1/eg/9.want
   A    proj1/eg/10.want
   A    proj1/eg/1
   A    proj1/eg/2
   A    proj1/eg/3
   A    proj1/eg/4
   A    proj1/eg/5
   A    proj1/eg/6
   A    proj1/eg/7
   A    proj1/eg/8
   A    proj1/eg/1.want
   A    proj1/eg/9
   A    proj1/eg/2.want
   A    proj1/eg/3.want
   A    proj1/eg/4.want
   A    proj1/eg/5.want
   A    proj1/eg/6.want
   A    proj1/eg/7.want
   A    proj1/lib.mk
   A    proj1/Makefile
   ```

4. Change directory to proj1
5. Edit the file ***Makefile*** and change the following lines:

   ```
   Group=11112222
   Proj=1a
   ```

   Make the Group the last four digits of the IDs of the two members of this group. And make the Proj the current project number (1).

If all the above works, you should be able to run the following:

```
cd proj1
make score
```

And get this output:

```
?? line 12: [tags  2 things]
?? line 26: [tags  3 boys 7 girls  ]
?? line 6: [tags  3 apples 10 oranges 16 pears]
?? line 5: [tag 2 things 7 *]
?? line 6: [tag 3 boys 7 *]
?? line 7: [tag 7 girls 13 +]
?? line 3: [tag 3 apple 7 *]
?? line 4: [tag 10 oranges 24 +]
?? line 5: [tag 16 pears 16 $]
   6 FAILED
   4 PASSED
```

Understanding the Test Engine

All your assignments will be marked by the same test engine. You actually have a copy of that test engine in ***proj1/Makefile***. It is therefore very important that you understand that test engine.

Inside a test engine

A test engine has tests to run and some expectation of the output of that run. The test engine smiles if the actual output is the same as the expected and frowns if they are different.

To make this all work, we need to

1. Define a test; e.g. "1"
2. Write a file showing what we expect from this test; e.g. "1.want"
3. Write something that runs "1" and checks it against "1.want".

That last step is generic; the same code can be used to test "1" and "2" etc. We saw some of that code before in pipe.

But the first two steps are different for every test. So we'll need a little more information on those.

Writing a new test

Tests are stored in proj1/eg/N (where N is a number). The code you downloaded above contains numerous input tests of the GRAPH tool. For example, you could have written the file eg/6) that contains these lines:

```
label mean IQ
left ticks  0 10 20 30 40
height 20
width  30
3 1 *
3 2 *
3 3 *
3 4 *
3 5 *
3 6 *
3 7 *
7 1 +
7 2 +
7 3 +
7 4 +
7 5 +
7 6 +
7 7 +
7 8 +
7 9 +
7 10 +
7 11 +
7 12 +
7 13 +
range 0 0 10 20
tags  3 boys 7 girls
```

"Run" a new input file

The file ***proj1/Makefile*** contains code that runs some file ***eg/N*** through GRAPH. Internally, that code calls

```
cat proj1/eg/N | gawk -f proj1/graph.awk
```

If you want to see how that works, look at ***proj1/lib.mk*** (which is included in the first line of ***proj1/Makefile***) but you don't need to. All you need to do is run

```
make X=N run
```

where "N" is one of the files in ***proj1/eg/N***. For example:

```
make X=3 run
```

generates:

```
---| 1 |-----------------------------------------

      |----------------------|
20    -                      |
      |                      |
      |         +            |
10    -            + +       |
      |       *              |
      |  * *                 |
2     *--|------|------------|
     1  2      5            10
```

Defining "run"'s expected output

Now what if you want to write your own test? There are two steps:

1. Create a new file e.g. ***proj1/eg/11*** (the name has to be an integer).
2. Create an expectation for that test. To do that, write a new test and when it is producing the output you like, run

   ```
   make X=11 cache
   ```

   This will catch the output of ***make X=11 run*** and place it in ***proj1/eg/11.want***

How to run a new test

Now, finally, you can run the new test:

```
make X=11
```

If the output matches ***proj1/eg/11.want***, then the system prints PASS. Else, you get a FAIL.

How to run all tests

Test engines are very useful when changing code. Once you have a couple of tests, then when ever you make changes, you can run the code to check that the old stuff still works.

The file ***proj1/Makefile*** contains code to run all the tests in ***proj1/eg/N***:

- ***make tests*** runs all the tests;
- ***make score*** runs the tests, then counts up the number of PASSes and FAILs (using the code introduces in the pipe and fi lterlecture.

What to do

Generate the files described in **bold**, below.

1. Show that you understand the set up instructions. Show that you can run the test code using

   ```
   cd cs310/proj1
   make tests > tests.out
   ```

2. Show that you understand GRAPH. Write some GRAPH input files ***eg/11, eg/12, eg/13, eg/14, eg/15***.
3. Show that you understand how to write tests. Create output files for ***eg/11.want, eg/12.want, eg/13.want, eg/14.want, eg/15.want***.
4. Show that you understand the test engine

```
cd cs310/proj1
make X=12  > test12.out
make tests > tests.out
```

5. Show that you understand GAWK. Read http://unbox.org/wisp/var/timm/09/310/lib/proj1a/GRAPH.awk and write a fi le ***gawk.txt*** listing all the GAWK features exercised by that file. You will need to write brief descriptions for the the following concepts:
   - Patterns, actions
   - next
   - BEGIN
   - END
   - Reguar expressions (explain "^[-+]?([0-9]+[.]?[0-9]*|[.][0-9]+)([eE][-+]?[0-9]+)?$"
   - All the GAWK built-ins used by GRAPH: sub, $0, $1, NR, NF, substr, split, length,int

   If you need help wih GAWK, see
6. Show that you understand GRAPH. Read http://unbox.org/wisp/var/timm/09/310/lib/proj1a/GRAPH.awk and write a fi le ***graph.txt*** listing all the GRAPH-specific functions written for this code. You will need to write brief descriptions for the the following functions:
   - expand, expnad1, frame, ticks, label,data, draw, xscale, yscale, plot, splot,saya