CSC 352, Fall 2015
Assignment 4
Due: Wednesday, September 23 at 23:59:59

Introduction

The `a2` write-up started with 3+ pages of various information about assignments. None of that is repeated here, but all of it applies this assignment, too.

You'll need to make an `a4` symlink, of course.

`CMD && CMD`

When working with Java it's easy to make a change and accidentally run the old version by forgetting to do `javac`. I often do this:

```
% javac x.java && java x
```

That compiles `x.java` and if there are no errors, it runs `java x`. After editing, an up-arrow (or `!javac`) compiles and runs again. I prefer the `&&` over `javac x.java; java x` because with `&&` a compilation error stops things but with a semicolon it always runs `java x`, sometimes rolling the error off the screen.

Problem 1. (3 points) `patlen.java`

A good exercise for understanding wildcards is to consider a pattern, like `???.[ch]`, and compute the minimum and maximum lengths of the entries that could be matched by the pattern. For this problem you are to write a program `patlen.java` that does that computation for you.

Here's how it works:

```
% java patlen
xyz
Pattern: xyz
Length: 3

??*
Pattern: ??*
Length: 2 to Infinity

???.[ch]
Pattern: ???.[ch]
Length: 5

[ab][cde][fghi]
Pattern: [ab][cde][fghi]
Length: 3

*x*y*z*
Pattern: *x*y*z*
Length: 3 to Infinity

*
```

```
Pattern: *
Length: 1 to Infinity

^D
%
```

We see that the user enters a line with a pattern that might or might not contain wildcards. `patlen` echoes the pattern and then outputs the length of entries that could be matched by the pattern. If the minimum and maximum is the same, like with "`xyz`", a single value is output. If there's a range of lengths, "*`N`* `to Infinity`" is output. (Yes, an entry name can't be infinitely long but we'll ignore that.)

`patlen` reads patterns and prints lengths until it reaches the end of standard input.

Assume the pattern doesn't contain whitespace or slashes. Assume the pattern is well-formed.

Hint: The computation is very simple—don't make a big problem out of this! For fun, my solution uses `Float.POSITIVE_INFINITY`.

Problem 2. (16 points) `lpp.java`

`a3`'s `clp.java` did just a little bit of command-line parsing. In this problem you're to write `lpp.java`, which does just a little bit of C-like preprocessing. It supports only two directives, `#include` and `#define`, and does macro expansion that's much more simple-minded than the real C preprocessor.

`lpp` assumes it has a single argument, a path to a file to preprocess. It reads the file line by line, handling any `#include` or `#define` directives it encounters, and writes the preprocessed lines to standard output.

Here's a simple example of `lpp` in operation:

```
% cat a4/lpp1.c
#define LOW -5
#define HIGH 10

int in_range(int i)
{
    return LOW <= i && i <= HIGH;
}
% java lpp a4/lpp1.c

int in_range(int i)
{
    return -5 <= i && i <= 10;
}
```

`lpp1.c` has two `#define` directives, for the macros `LOW` and `HIGH`. In the output of `java lpp a4/lpp1.c` we see that the occurrences of `LOW` and `HIGH` in `lpp1.c` have been replaced with `-5` and `10`, respectively.

`lpp` is not nearly as smart as the real C preprocessor. The real C preprocessor only does macro replacement on full identifiers but `lpp` blindly replaces every occurrence of the string `"LOW"` with `"-5"` and every occurrence of `"HIGH"` with `"10"`. Observe:

```
% cat a4/lpp2.c
#define LOW -5
#define HIGH 10

IT IS HIGHER THAN NEEDED! TELL THEM TO "LOOK OUT BELOW!" WHEN
LOWERING IT.
```

Here's what `lpp` does with that file. Replaced text is shown bold and underlined.

```
% java lpp a4/lpp2.c

IT IS 10ER THAN NEEDED! TELL THEM TO "LOOK OUT BE-5!" WHEN
-5ERING IT.
```

Notice that much like the real C preprocessor, `lpp` knows nothing about C! The text in `lpp2.c` is surely not valid C code but `lpp` is happy to preprocess it! Some examples below use simple text rather than C code to help focus on the mechanics of macro expansion.

Implementing macro expansion

For each input line you'll obviously need to loop through all the macros thus far and do replacements. Use `String.replace()` to replace all occurrences of a macro with its expansion, like `line.replace("LOW","-5")`. **Calling `replace()` is simple enough but there's a twist: an expansion might contain strings that themselves need to be replaced.**

Consider this example:

```
% cat a4/lpp3.c
#define zzz the end
#define almost zzz of zzz
#define start almost

Near start
% java lpp a4/lpp3.c

Near the end of the end
```

To make `lpp3.c` work, you'll need to do "rescanning". The idea of rescanning is simple:

(1) *If a pass through the macros produces any change in the line, make another pass through macros.*

(2) *Repeat until a pass through the macros fails to produce any change in the line.*

Here are the changes that the line `"Near start"` undergoes:

```
Near start
Near almost
Near zzz of zzz
Near the end of the end
```

Note that the ordering of those three `#define`s makes no difference in the final result. Here's another permutation:

```
% cat a4/lpp3a.c
#define start almost
#define zzz the end
#define almost zzz of zzz

Near start
% java lpp a4/lpp3a.c

Near the end of the end
```

There may be cases where the ordering of macro expansion makes a difference but as of press time I don't have any to cite. We won't test with any such cases.

An implication of rescanning is that you can create an infinite loop with the macros. Here's a trivial case:

```
#define a b
#define b a
```

A line with `"aaa"` will turn into `"bbb"`, and then `"bbb"` will turn into `"aaa"`, an infinitely repeating cycle. Don't worry about this possibility! If the user makes a loop with their macros, then `lpp` will loop!

You're encouraged to try to work out line processing on your own but you can find pseudo-code for it in `a4/lpp-processLine`.

Representing macros

I recommend representing macros as key/value entries in a `HashMap<String,String>`. One way to think of a `HashMap<String,String>` is that it is like an `ArrayList` of strings that can be indexed by strings instead of integers.

Following below is textual copy/paste from interaction with www.javarepl.com that shows a `HashMap<String,String>` in operation. Various non-useful output has been elided.

First, let's make a `HashMap<String,String>` named h:

```
java> h = new HashMap<String,String>();
java.util.HashMap h = {}
```

Let's use `HashMap`'s `put` method to associate the key `"LOW"` with the value `"-5"`, and the key `"HIGH"` with the value `"10"`.

```
java> h.put("LOW", "-5")

java> h.put("HIGH", "10")
```

We can see that two key/value pairs are now in the HashMap:

```
java> h
java.util.HashMap h = {HIGH=10, LOW=-5}
```

Obviously, we can use a key to represent a macro name. The value associated with the key is the macro's expansion.

With that macro name/macro expansion mindset for keys and values, let's fetch the expansions associated

with the macros `LOW` and `HIGH`:

```
java> h.get("LOW")
java.lang.String res3 = "-5"

java> h.get("HIGH")
java.lang.String res4 = "10"
```

`HashMap`'s `keySet` method produces a set of the keys:

```
java> h.keySet()
java.util.HashMap.KeySet res5 = [HIGH, LOW]
```

It seems that `for`-loops don't work on `javarepl.com`, so let's switch to a Java program that starts with the code above but then loops through the macros and prints the expansion for each.

```
% cat a4/lpphashmap.java
import java.io.*;
import java.util.*;

public class lpphashmap
{
    public static void main(String args[])
    {

        HashMap<String,String> macros =
           new HashMap<String,String>();

        macros.put("LOW", "-5");
        macros.put("HIGH", "10");

        for (String macroName: macros.keySet())
        {
            String macroExpansion = macros.get(macroName);
            System.out.format("macro %s expands to %s\n",
                macroName, macroExpansion);
        }
    }
}

% javac lpphashmap.java && java lpphashmap
macro HIGH expands to 10
macro LOW expands to -5
```

Handling `#includes`

The hard part of `lpp` is handling `#defines` but `lpp` also supports `#include`. Here's an example that involves several files. This is the "top level" file (we'll eventually run `java lpp a4/lpp-inc1.c`.)

```
% cat a4/lpp-inc1.c
#include "a4/lpp-limits.h"
#include "a4/lpp-funcs.h"

int main()
{
   printf("Version: %s\n", VERSION);
```

```
   printf("the range of an int is %d to %d\n", INT_MIN,
INT_MAX);
}
```

We can see that `a4/lpp-inc1.c` includes these two files:

```
% cat a4/lpp-limits.h
// from lpp-limits.h
#include "a4/lpp-version.h"
#define INT_MAX 2147483647
#define INT_MIN (-INT_MAX - 1)

% cat a4/lpp-funcs.h
extern int printf (const char *format, ...);
extern int atoi (const char *s);
```

In turn, `a4/lpp-limits.h` includes `a4/lpp-version.h`:

```
% cat a4/lpp-version.h
/* current version */
#define VERSION "C11 on lectura"
```

Let's preprocess the top-level file, `a4/lpp-inc1.c`:

```
% java lpp a4/lpp-inc1.c
// from lpp-limits.h
/* current version */
extern int printf (const char *format, ...);
extern int atoi (const char *s);

int main()
{
   printf("Version: %s\n", "C11 on lectura");
   printf("the range of an int is %d to %d\n", (-2147483647 -
1), 2147483647);
}
```

We can see that the only remaining trace of `a4/lpp-limits.h` is its first-line comment. The other three lines were consumed by `lpp`. The same is true of `a4/lpp-version.h` that was included by `a4/lpp-limits.h`—only its first-line comment is left. The two lines in `a4/lpp-funcs.h` went through unchanged.

Finally, in `main` above, we see the effects of the `#define`s: the macros for `VERSION`, `INT_MIN`, and `INT_MAX` have been expanded.

If an included file doesn't exist, "*`FILENAME`*`: can't open`" is written to standard error, `System.exit(1)` is called immediately, **and what's written to standard output is undefined**. Example:

```
% cat a4/lpp-inc2.c
// a line here
#include "abc.xyz"
// and a line here...
```

```
% java lpp a4/lpp-inc2.c >/dev/null
abc.xyz: can't open

% echo $?
1
```

Note that standard output is redirected to `/dev/null` to emphasize that we don't care about it. We only care that the `"can't open"` line went to standard error and that `System.exit(1)` was called, which is demonstrated by echoing the bash special variable `$?`, which holds the exit code of the last-run command.

Simplifying assumptions

- `lpp` is given one argument: a file to preprocess.
- The directives `#include` and `#define` must be at the start of a line. Use `String.startsWith()` to look for them.
- Assume that `#include` is followed immediately by a space, a double quote, a pathname, and a (closing) double quote. The closing double quote is the last character on the line.
- Assume that `#define` is followed immediately by a space, a sequence of one or more alphanumeric characters and underscores, and space. Everything following the space through the end of the line is the expansion text for the macro.
- Assume a macro is never redefined. For example, you won't see this:

```
#define a x
#define a y
```

- The student set of tests for grading `lpp` will "freeze" at 23:00 on March 19—no tests will be added after that time. That frozen student set will be used as the grading set. If you pass all the tests in that student set, you are guaranteed 100% on this problem. Needless to say, any solutions that have been tailored for the tests, perhaps by looking for `"LOW"`, `"HIGH"`, etc., will receive a zero.
- Don't pay much attention to the C code used above; focus on the textual transformations instead.
- Don't use the real C preprocessor as a reference version of `lpp`; their behaviors differ in various ways.

Beat-the-Coach time: 54 minutes, 36 seconds. No IDE.

Problem 3. `questions.txt`

`a4/questions.txt` is a plain text file with a number of free-response questions. Copy the file into your directory and edit your answers into it, leaving the questions intact. Add your answers after the `"Answer:"` lines. Answers may be multi-line. Per-problem points are specified in the file.

If you're unsure about the format for any of your answers, mail it to `352f15` and we'll take a look!

Problem 4. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with `"Hours:"`.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a4/turnin` to submit your work. It creates a time-stamped "tar file" of your work. You can run it as often as you want. We'll grade your final submission.

`a4/turnin -l` shows your submissions.

To give you an idea about the sizes of my solutions, here's what I see as of press time.

```
% wc $(cat a4/delivs | grep java)
  45  113 1259 patlen.java
  83  217 2703 lpp.java
 128  330 3962 total
```

There are no comments in my code.

Miscellaneous

`a4/tester` is not in place as of press time but will be operational before Friday.

This assignment is based on the material on UNIX slides 1-177 and C slides 1-40.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem would correspond to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.)

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that a student who has only taken CSC 127A and 127B but done well in them and has completed the previous assignments will need 8-10 hours to complete this assignment.

Keep in mind the point value of each problem; don't invest an inordinate amount of time in a problem or become incredibly frustrated before you ask for a hint or help. Remember that the purpose of the assignments is to build understanding of the course material by applying it to solve problems. If you reach the eight-hour mark, regardless of whether you have specific questions, it's probably time to touch base with us. Give us a chance to speed you up! **Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.**

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, and more. (See the syllabus for the details.)