CS 61BL project 2

Goals

The goals for this project are to solidify your skills at manipulating trees and working with tree recursion, and to give you practice constructing and testing a relatively large program.

Submit a team solution as `proj2` by 10pm on Sunday, July 28. Submit individual evaluations of team members by Sunday as well, in the same way you did for project 1. Your team gets 24 "grace hours" to use toward submitting projects late, minus whatever grace hours you used on project 1.

T.a.s will interview your team on July 18 and July 23 to assess your progress and to address any problems relating to your team activities. All members of your team must attend these meetings. (The meetings will earn homework points.)

High-level project description

You are to write a proof checker for propositional logic. The user will type steps of a proof; your program will examine each step as it is input, and either verify that the step makes sense or complain if the step is invalid. A separate document provides more background on propositional logic and proofs.

The user interface

Each input line will consist either of the word "print" appearing alone on the line (except for white space), or of a *reason*, followed by blank space, followed by an *expression*. A reason is one of the following:

- `show`
- `assume`
- one of the words `mp`, `mt`, or `co` followed by blank space, followed by a line number and blank space and another line number
- one of the words `ic` or `repeat` followed by blank space and a line number
- the name of a *theorem* (see below)

The words "mp", "mt", "co", and "ic", abbreviate the terms "modus ponens", "modus tollens", "contradiction", and "implication construction". See the background section for more details.

A line number is a sequence of positive integers alternating with periods, for example, 1.3.10, 2, or 3.5.8.1. Line numbers given in proof steps must label lines already used. An additional restriction is that an inference is not allowed to depend on the *interior* steps of a previous "show" sequence. A legal line number reference matches the current line number everywhere but in the last place, which must be less than the corresponding component of the current line number. For example, if the current line number is 3.2.4, any of the following lines may be referenced: 1, 2, 3.1, 3.2.1, 3.2.2, or 3.2.3.

Expressions have the following format.

- a variable (a lower-case letter) is an expression;
- if *`E`* is an expression, then `~`*`E`* is an expression;
- if *`E1`* and *`E2`* are expressions, then `(`*`E1`*`&`*`E2`*`)` and `(`*`E1`*`|`*`E2`*`)` are expressions;
- if *`E1`* and *`E2`* are expressions, then `(`*`E1`*`=>`*`E2`*`)` is an expression.

An expression contains no spaces. An expression is fully parenthesized; that is, parentheses enclose each use of the binary operators `&`, `|`, and `=>`, and appear nowhere else in the expression.

Steps of a proof may also be instances of *theorems*. The name of a file of theorems may be specified as a command line argument. Each line in the file contains the name of a theorem (you choose the names), followed by blank space, followed by an expression. A theorem is used by giving its name (which should not be "print", "show", "assume", "mp", "mt", "co", "ic", or "repeat"), followed by an expression that is consistent with the theorem's expression. For example, suppose that the theorem file contains the following theorems for working with the `&` and `~` operators ("dn" stands for "double negative"):

```
and1 ((x&y)=>x)     and2 ((x&y)=>y)     dn (~~x=>x)
```

An application of the `and1` theorem would be

```
and1 (((a|b)&~c)=>(a|b))
```

which applies the theorem using `(a|b)` for `x` and `~c` for `y`.

Theorems involving `&`, `|`, and `~` will be useful for thoroughly testing your program.

The line number arguments for the `mp`, `mt`, and `co` may appear in either order. Suppose, for example, that line 5 contains `((p&q)=>(r|s))` and line 7.2.3 contains `(p&q)`. Then either of the steps

```
mp 5 7.2.3 (r|s)     mp 7.2.3 5 (r|s)
```

may be used to extend the proof, provided line 7.2.3 is accessible from the current line.

A subproof is begun with the `show` reason. The line number of the step immediately following is the line number of the `show` step with a ".1" appended. Subsequent lines continue with ".2", ".3", and so on, until either another`show` is given (which nests the numbering one more level) or a step is given, on *the same level as* the `show`, in which the inferred expression is identical to the `show`'s operand. In the latter case, the subsidiary derivation is complete; line numbering then continues with the number following that of the `show`.

The first `show`, supplied by the user on line 1, is handled specially. The line number of the line following the first `show` is 2, not 1.1.

Assumptions—uses of `assume`—may appear only immediately after a `show`. After the step `show E1=>E2`, the step `assume E1` may appear. After the step `show E`, the step `assume ~E` may appear.

A line containing only the word "print" causes the program to print each line of the proof in order, with each line preceded by its line number and a tab character. It does not extend the proof or have any other effect.

Here's how the proofs in the background section would appear.

```
1  show (q=>q) 2  assume q 3  ic 2 (q=>q)  1  show (p=>(~p=>q)) 2  assume p 3  show
(~p=>q) 3.1  assume ~p 3.2  co 2 3.1 (~p=>q) 4  ic 3 (p=>(~p=>q))  1  show
(((p=>q)=>q)=>((q=>p)=>p)) 2  assume ((p=>q)=>q) 3  show ((q=>p)=>p) 3.1  assume
(q=>p) 3.2  show p 3.2.1  assume ~p 3.2.2  mt 3.2.1 3.1 ~q 3.2.3  mt 2 3.2.2 ~(p=>q)
3.2.4  show (p=>q) 3.2.4.1  assume p 3.2.4.2  co 3.2.4.1 3.2.1 (p=>q) 3.2.5  co
3.2.4 3.2.3 p 3.3  ic 3.2 ((q=>p)=>p) 4  ic 3 (((p=>q)=>q)=>((q=>p)=>p))
```

In each of these examples, the program prints the line number and the user types the `show`, `assume`, or an inference.

Here's one more that uses `repeat`. Note that step 2 is something of a nonsense step.

```
1 show (p=>p) 2 show (p=>p) 2.1 assume p 2.2 ic 2.1 (p=>p) 3 repeat 2 (p=>p)
```

Also online is a proof that contains a larger example, with a theorem use.

Detailed project description

Given below is a framework for the project program. It's also available in `~cs61bl/code/proj2/ProofChecker.java`. Don't change anything in the framework.

```
import java.io.*; import java.util.*;  public class ProofChecker {      private
InputSource lines;             // source for steps in the proof     private
TheoremSet myTheorems; // theorem repository     public static boolean iAmDebugging
= false;          // The first command-line argument, if specified, is the name of
// a file of theorems (name/expression pairs).     // The second command-line
argument, if specified, is the name of     // a file from which to read steps in the
proof.    // If no theorem file name is specified, proof steps are read     // from
standard input.         public ProofChecker (String [ ] args) {         // Set up
input of proof steps.        if (args.length > 1) {            lines = new
InputSource (args[1]);         } else {            lines = new InputSource ( );
}                  myTheorems = new TheoremSet ( );         if (args.length > 0) {
InputSource theoremsIn = new InputSource (args[0]);           while (true) {
// Read a line from the theorem file.                 String line =
theoremsIn.readLine ( );                 if (line == null) {
return;                 }                 // Turn it into a theorem and put it into
the theorem collection.                 Scanner thmScanner = new Scanner (line);
String theoremName = thmScanner.next ( );                 Expression theorem = null;
try {                     theorem = new Expression (thmScanner.next ( ));
myTheorems.put (theoremName, theorem);              } catch (Exception e) {
System.err.println ("*** bad theorem file: " + args[0]);
System.exit (1);                 }             }         }         }      public
static void main (String [ ] args) {         if (args.length > 2) {
System.err.println ("too many arguments");           System.exit (1);         }
ProofChecker checker = new ProofChecker (args);        Proof soFar = new Proof
(checker.myTheorems);                     boolean done = false;         while
(!done) {             System.out.print (soFar.nextLineNumber ( ) + "\t");
try {                 String line = checker.lines.readLine ( );
soFar.extendProof (line);                done = soFar.isComplete ( );           }
catch (IllegalLineException e) {                System.out.println
(e.getMessage());            } catch (IllegalInferenceException e) {
System.out.println (e.getMessage());             }         }     } }
```

The framework includes references to a `Proof` class that keeps track of the steps of the proof being worked on. The directory `~cs61bl/code/proj2/Proof.java` contains an abbreviated version of this class. Methods of the`Proof` class include the following:

- a constructor that takes a `TheoremSet` as argument,
- `nextLineNumber ( )`, which returns the number of the next line of the proof to be filled in,
- `extendProof (String)`, which tries to extend the proof with the given input line,
- `toString ( )`, which returns a printable version of the legal proof steps typed so far, and
- `isComplete ( )`, which returns `true` if the most recent line matches the expression given in the outermost `show` step and is not part of a subproof and returns `false` otherwise.

Here is how the program will work.

1. The `main` method creates a `ProofChecker` object, which sets up an `InputSource` of lines and reads theorems from a file if specified.
2. It then creates a `Proof` object that will store the proof being completed.
3. It enters a loop:
    a. It asks the `Proof` what its current line number is, and prints it.

b. It asks the `InputSource` for a line, then sends that line to `Proof.extendProof`. Either it will be a valid extension to the proof, or `extendProof` will throw an exception.
c. Finally, it will check if the proof is completed, leaving the loop if so.

More code details

The `~cs61bl/code/proj2` directory also includes the `InputSource` class, two exception classes, and abbreviated versions of classes to represent expressions and collections of theorems.

You may assume that the file of theorems named in a command-line argument is free of errors. However, you should thoroughly analyze each line typed *by the user*, and it should be *impossible* for the user to crash your program (other than by hitting control-C).
The `extendProof` method should throw one of two exceptions upon encountering an illegal proof step:

- `IllegalLineException` is thrown for a line whose *syntax* is incorrect. Input that would cause this exception are lines with the wrong number of arguments, with an invalid reason or expression, or with garbage line numbers.
- `IllegalInferenceException` is thrown for a line whose *semantics* or meaning is incorrect. Examples of semantic errors are any errors for which you need to compare the input with the proof in order to determine its validity, for example, a use of modus ponens with an expression that's not an implication, or an incorrect instantiation of a theorem.

The error messages with which you initialize `Exception` objects should be as informative as possible about the error that was caught.

Another component of your program will be a pattern matcher for expressions. (You may recall the pattern matcher used in the implementation of the query language in CS 61A.) The four inference rules listed above require a limited pattern matching capability. Use of a theorem requires a more extensive facility that's given the theorem as a pattern and an expression to infer using the theorem. For example, the theorem

```
(((p=>q)&(~p=>q))=>q)
```

matches any expression E whose top-level operator is `=>`, whose left child's operator is &, whose left grandchildren are implications, and in which all the occurrences of `p` in the theorem match the same subexpression of E and all the occurrences of `p` in the theorem match the same subexpression of E. An example match, with `p` in the theorem matching `(r|s)` and `q` in the theorem matching `(x|~y)`:

```
((((r|s)=>(x|~y))&(~(r|s)=>(x|~y)))=>(x|~y))
```

You may find the `java.util.Scanner` class useful in parsing input lines. It implements the `Iterator<String>` interface, and provides a convenient way to split a string into fields. It's used in the `ProofChecker.java`framework.

Miscellaneous requirements

Detailed design, and the coding and testing of your solution to this assignment should be the work of your team and no one else. (Upcoming discussion homework will involve various aspects of high-level design and development.) Get started early; our solution is several hundred lines of code.

You may add as many other classes as you want to those provided. (A good heuristic is to represent each *noun* in the problem description with a class.) All data fields and methods within each class must have the proper public/private/protected/package qualifier. In particular, you are not allowed to make things public that would let a non-inheriting user corrupt your data structures (even if none of your own code would do this).

For each nontrivial class you include in the program other than `ProofChecker`, provide a JUnit class named `...Test` to generate evidence that your methods work correctly. These will include at least `ProofTest.java` and`ExpressionTest.java`.

Also provide a writeup, in a file named `readme.txt` or `readme.pdf`, a description of the rationale for your tests, what bugs each test would direct you to if it failed, and why your tests provide sufficient evidence of the absence of bugs in your program. (Please convert writeups in other formats to PDF format.) Refer to previous discussion activities on the topics of completeness and organization of tests in your writeup. Include in your writeup a brief description of what each of your team members contributed to the solution.

If your program produces any output other than what's already produced in the program framework, surround the output statement by checks of the `iAmDebugging` variable in the `ProofChecker` class. *Do not* remove the debugging output statements from your program before submission. Your submission will be evaluated with `iAmDebugging` set to false.

Your program should exhibit good style and organization. Choose good names for variables, and comment each of your methods appropriately. Indent code to show its structure in a consistent way. Limit the length of your methods to a screenful or so.

Milestones

In lab activities on July 18, your group will meet with course staff to evaluate your plan for completing the project. Staff will ask you who will be doing what. The individual group members will elaborate on their chosen tasks, predicting what the hard parts will be and describing how much they will have completed and tested by the next checkpoint. A small number of homework points will be awarded for this activity. You'll earn full credit for a reasonable plan.

In lab on July 23, you will brief your t.a. on your progress: what you each have finished, what you failed to complete that you promised to finish in the previous week (and why), and what's left. A somewhat larger number of homework points will be awarded for this activity, based on how far along you are and on whether or not you completed the tasks that you promised to finish on the 18th.

You don't have to do everything at once. Here are some suggestions for subtasks that you may wish to consider:

- one kind of error checking, say, checking the syntax of a line or detecting wrong line numbers;
- parsing an expression;
- at least two of checking `isComplete`, managing line numbers, and implementing `toString`;
- making an inference (once you can handle modus ponens, the rest are straightforward);
- handling theorems.

Submission requirements

Submit a solution to this project by 10pm on Sunday, July 28. Do this via the command

```
    submit proj2
```

Your submitted files should include the following:

```
    Proof.java     ProofTest.java     Expression.java     ExpressionTest.java
readme.{txt,pdf}
```

plus whatever other source files and test files you choose to submit.

Submit one project solution per team. Then individually, submit partnership ratings as you did with project 1.

The various components of this project will contribute to its grade roughly as follows.

- error handling: 20%
- legal expression parsing: 20%
- application of inference rules: 20%

- theorem use: 20%
- testing writeup and files: 20%

Your project grade will also depend on ratings by your group members. A weighting factor will be computed by dividing your collaboration ratings by the average of the ratings in your team; the weighting factor will then be multiplied by the raw project grade to determine your grade for the project.

More Information

- Before asking the course staff questions, please make sure you have read this FAQ page.
- If you'd like to see some proofs done via truth tables, check out this tutorial page, courtesy of the Spring 2011 CS61BL staff.
- We've also provided a set of sample proofs to help you understand how your program is supposed to function.