analogy readme / implementation notes
Gordon Krieger, April 2016

COMPILATION

Note: this does not compile on Trottier Ubuntu machines. Further, the theorem prover used in this implementation, Prover9, does not compile at Trottier, either. (I tried the current version and two older ones, using the included makefile in all cases). The issue appears to be missing libraries. Since the prover itself doesn't compile, I didn't see the point of trying to beat my own code into shape to get it to run at Trottier; performance without the prover is possible, but largely uninteresting. The code *should* work on any POSIX system, but obviously that claim comes with a lot of caveats. This was developed on a MacBook Pro, but contains no OSX-specific code. Requires C++11, which will be present on most machines with a gcc compiler. No special libraries are required, although you do need to download the prover.

To compile:

> (step 0: *Ask me to come in compile it for you. I am happy to do that for anyone connected to the project or the course. Alternately, follow these steps yourself:*)
>
> 1: Go to `https://github.com/gsfk/analogy/` and click on "Download ZIP" to download all files. I can also send an archive by email.
>
> 2: Download the theorem prover, Prover9:
> Go to `https://www.cs.unm.edu/~mccune/mace4/download/` and select the most recent version.
>
> 3: Compile Prover9 using the instructions found on the download site.
>
> 4: Find your *absolute* path to the Prover9 executable. The simplest way is probably to navigate your way there in Terminal, then type the command `pwd` to print the full path, then copy it. It should be something along the lines of `/Users/me/Documents/stuff/junk/LADR-2009-11A/bin/`
>
> 5: The implementation needs to know where your copy of Prover9 is. You need to paste the full path for your prover into the code. From my code, open the file `prover.cpp` and change line 19:
>
> ```
> int outcome = system("/...your..path...here..../prover9 -f analogy_prover_file.in > prover_temp.out");
> ```
>
> Replace the dummy path name with your path, taking care not to remove the leading quotation mark, and that the space after "prover9" remains. I apologize for this step; the most I would normally expect would be for users to paste their path into some `#define` statement in a header file, but OSX does not play nice with this approach, so it was impossible to implement or test on my machine. Another obvious option, writing a second app to do all this work for you, was not something I had time to develop.
>
> 6: In Terminal, find your way to the folder containing my code, then type `make` or `make all` to compile. If something goes wrong (such as screwing up step 5), use the command `make clean` before attempting to recompile.

Again, I happy to come by and do this for you.

OPERATION

When given a text file as representing a structure as input, `analogy` produces a list of true formulas for that structure, followed by a minimal subset.

More interesting output is produced by giving two files as input. The formulas for both input structures are given and compared, along with their respective minimal sets.

Several example input files are included in the submission. Example operations:

`./analogy 3ng.txt`
gives formulas and axioms for the "3-no-group" example from Dirk's paper "Two Ways of Analogy."

`./analogy mod3.txt 3ng.txt`
compares the 3-no-group with a group of the same order

`./analogy g10acyclic.txt g10cyclic.txt`
compares the cyclic group of order 10 with the acyclic group

`./analogy 6.2.txt mod5.txt`
compares an abelian (commutative) group with a non-abelian one.

In all cases you can add "`-noprover`" as a final argument to disconnect the theorem prover, but expect less than exciting output.

`analogy` will create input and output files from the prover in your current working directory, you can safely ignore these. For best results run the program from the folder where it resides.

MAKING YOUR OWN FILES

Example input files should be enough to demonstrate the capabilities of the implementation. The largest file is order 19; I have not focused on generating larger input since the number of formulas discovered is largely independent of the size of the input.

Writing and debugging a good parser is a project in itself, and is far from the intellectual centre of the project. So in order to avoid spending too much time on the parser, I've generally held to Dirk's suggestion to assume that input is in the correct format. There is some small amount of error checking and some accommodation for comments. The parser works on all the example input but is otherwise brittle. Please don't try to break the parser!

Input files should follow the format of the example files:

```
Elements:  0, 1, 2

Relation:  *

Facts:
(0,0,0)
(0,1,1)
(0,2,2)
(1,0,1)
(1,1,2)
(1,2,0)
(2,0,2)
(2,1,0)
(2,2,1)
end
```

Note the following limitations:

Elements should be separated by a comma and a space. Spaces are not necessary for facts.

The name of the relation should be a single character.

Elements need not be numbers but they are limited to single characters. So for numbers greater than 9, use letters, both capital and lowercase. Beyond that, any standard ascii character will do. The parser was written first, when I was expecting a much lower limit on input size. The relation name is also restricted to a single character.

The implementation currently works only for binary operators, so all facts should contain 3 elements.

All example input uses "$*$" as the name for the binary operator. Any single character can be accepted as a relation name, but do not try to compare axioms for input with different relation names.