CS 485 Compilers (Sp25)

PA1: Rosetta Stone

DUE MONDAY, 3 FEBRUARY 2025, 11:59PM AOE.

The Rosetta Stone aided linguistic understanding by providing the same text in three different languages. In this project you will implement the same simple program in four separate languages. Each of your implementations will have exactly the same interface, will otherwise adhere to the same specification, and should behave exactly the same way.

You must choose one language from each of the following buckets for your four implementations:

- *Bucket 1: Languages You Already Know*: Java, C, C++, or Python3
- *Bucket 2: Languages With An Unusual Type System*: Kotlin, Rust, or Scala
- *Bucket 3: Functional Languages*: OCaml or Haskell
- *Bucket 4: Project Language*: Classroom Object-Oriented Language (COOL)

Each bucket is intended to challenge you in a different way. *Bucket 1* gives you a chance to implement the solution in a language you're already comfortable in, so that you can focus on the problem. *Bucket 2* challenges you to redo that implementation in a language that shares some similarities with languages that you've seen before, but which also includes some features you might be unfamiliar with. In particular, all *Bucket 2* languages support programming in a structured imperative style, even if they permit functional programming. *Bucket 3* challenges you to redo the implementation in a true functional programming language. Finally, *Bucket 4* will familiarize you with the course project language COOL ("Classroom Object-Oriented Language"). All subsequent assignments in this course (PA2-PA4) involve implementing parts of a COOL compiler, so you'll have an easier time with them if you've used the language a bit. COOL itself is also a bit tricky to program in: it intentionally omits many convenience features of higher-level languages to simplify the compiler implementation process, but that can make it unwieldy for writing "real" programs.

For this assignment, you must work **alone**. Subsequent assignments will allow you (optionally) to work in pairs.

Specification

Your program must take in a list of dependent tasks and either output a valid order in which to perform them or the single word `cycle`.

Your program will accept a number of lines of textual input (via standard input). There are no command-line arguments — you must always read from standard input. Do not open a named file. Instead, always read from standard input.

That text input will contain a non-zero but even number of lines. Every two lines represent a *pair* of tasks. The first line gives the name of a task, the second line gives the name of a task that it depends on. This text input is also called the *task list*.

The task list will contain only standard ASCII characters (no UTF/8 Unicode or special accents). The goal is to test programming and program language concepts, not your internationalization abilities.

Each task name starts at the beginning of the line and extends all the way up to (but not including) the end of that line. So the newline or carriage return characters `\r` or `\n` are not part of the task name. Each task name is at most 60 characters long. (This limit is to make any C implementation easier. Most languages support longer strings natively, and can thus ignore this length limit.)

Example task list:

```
learn C
read the C tutorial
do PA1
learn C
```

The interpretation is that in order to `learn C` one must first `read the C tutorial` and that in order to `do PA1` one must first `learn C`. Desired output for this example:

```
read the C tutorial
learn C
do PA1
```

If the task list containts a *cycle* of any size, your program should output exactly and only the word `cycle`. Example cyclic input:

```
get a job
have experience
have experience
work on a job
```

```
work on a job
get a job
```

Even if the task list contains a few non-cyclic parts, any single cycle forces you to output only the word `cycle`.

Always output to standard output only. Do not write anything to stderr.

There is no fixed limit on the number of lines in the task list (although it is not zero and it is even).

Two tasks with the same name are really just the same task. Use standard string equality.

Duplicated pairs of tasks are not allowed. For example:

```
learn C
read the C tutorial
do PA1
learn C
learn C
read the C tutorial
```

... that task list is not valid input because the pair `learn C` / `read the C tutorial` appears twice. Program behavior if the task list contains a duplicate pair is undefined. You will not be tested on it.

Your program may **not** cause any other file I/O to be performed, such as creating a temporary file to keep track of some intermediate sorting results or writing to stderr (or even causing the interpreter to write a warning to stderr). You do not need any such temporary files or stderr-printing to solve this problem.

Choosing Among Unconstrained Tasks

If there are multiple outstanding unconstrained tasks, your program should output them in ascending ASCII alphabetical order. That is, if you ever have two or more tasks, each of which has no remaining dependencies, output the one that comes first ASCII-alphabetically. (This constraint makes your program deterministic; for any given input there is only one correct output.) Example:

```
learn C
understand C pointers
learn C
read the C tutorial
```

```
do PA1
learn C
```

Because r comes before u, your output should be:

```
read the C tutorial
understand C pointers
learn C
do PA1
```

To put it another way, consider this task list:

```
B
A
C
D
C
E
```

Which yields a dependency graph like this:

```
A  D E
|  \ /
B   C
```

The proper ordering for this set of tasks is `A B D E C`. Note that `B` comes before `D` and `E`, even though `B` depends on `A`. This is because, once `A` is finished, `B` is free to go and it comes first alphabetically. You may want to consider this requirement when you pick your sorting algorithm. Given this requirement the answer `A D E B C` is incorrect and will receive no credit.

Resources

Some resources are available to help you:

- pa1-hint.zip provides concrete implementations of "task list reversal" (a similar, but simpler, problem) in a wide collection of languages, including many of those in our buckets. You might use this resource to compare how to accomplish various tasks in different languages.

- pa1-testcases.zip includes a number of test inputs and expected outputs so that you can test your programs before submitting.
- pa1-testcases-unix.zip as above, but with unix linefeed formatting in all text files for use in Mac OS X, Ubuntu, or another Linux.

Commentary

This problem is just topological sort not-so-cleverly disguised. Feel free to look up how to do toposort on the internet or ask your favorite generative AI tool to explain it to you (but remember that you must turn in your own work; you may not copy someone else's code and claim it as your own).

Take a look at the files in pa1-hint.zip. You could do worse than using them as starting points.

If you're having trouble writing anything reasonable in Cool, don't forget to look at the other example Cool programs.

Building and maintaining an explicit graph structure is probably overkill.

Video Guides

This assignment is a mildly-modified version of Wes Weimer's similar assignment (the only differences are in language choices). Wes has kindly prepared a number of video guides to help students get started in various languages:

- Python
- OCaml
- COOL (object-oriented style, long)
- COOL (imperative style, short)

Reminder: You can watch YouTube videos at 1.5x speed with full audio.

Checkpoints

This assignment has two checkpoints before the final due date. At the first checkpoint, you must turn in your *Bucket 1* implementation. At the second checkpoint, you must turn in any one implementation from a bucket other than *Bucket 1*. At the final due date, you need to turn in all four implementations. This structure is designed to encourage you to steadily work on the assignment, rather than trying to do all four implementations in a rush at the end. It also separates the task into three stages, each with a different challenge: (1) before the first checkpoint, the challenge is *Can I write this program at all?*, even in a familiar language; (2) between the first and second checkpoint, the challenge is *Can I write this program in some other language?*; (3) after the second checkpoint,