Take-Home Problem: Semantic Versioning

Your challenge is to implement Semantic Versioning, as specified by semver.org.

Your Challenge

In the language of your choice, design and implement a module that provides the following functionality:

- A convenient data type for semantic versions, in which major, minor, patch, pre-release, and metadata can all be separately accessed.
- A way to parse semver strings (the strings described in the specification into your semver data type.
- A way to convert your semver data type into semver strings.
- Comparison functions or operators that implement precedence, as described in the specification. In particular, two versions should be equal if and only if they have equal precedence.

Further, we will automatically test your program by passing it text on standard input, and seeing if it produces the expected text on standard output, according to the following specification:

Input/output format

Every *valid* line of input will contain, in order:

- a valid Semantic Version string
- one or more whitespace characters
- another valid Semantic Version string
- zero or more whitespace characters

For every line of input, your program should produce output as follows:

- If the line is empty, or contains only whitespace, ignore the line.
- If the line is invalid and contains a non-whitespace character, output a line reading 'invalid'.
- If the line is valid, compare the two versions according to the Semantic Versioning precedence specification.
  - If the left version is earlier the right version, output a line reading 'before'.
  - If the left version is later than the right version, output a line reading 'after'.
  - If the left and right versions have the same precedence, output a line reading 'equal'.

For example, given the following input:

```
1.3.6 1.4.2
1.7.9 1.3.5 0.0.2
4.2.3-beta    4.2.3-alpha
1.6 1.6.3
```

Your program should yield the following output:

```
before
invalid
after
invalid
```

Some Rules

- Use whatever development tools and languages you prefer.
- Please *do* refer to the Semantic Versioning specification, but do *not* refer to other implementations.

Artifacts to Submit

We'll be most interested in the following artifacts of your code:

- The code itself
- The setup and invocation instructions so that we can run your code on a Unix system
- Your tests for your code
- Whatever documentation will help us to:
  - Read and understand your code
  - See how to use your code in a larger system
  - Run *your* tests

What We'll be Looking For

We'll appraise your submission on the following general guidelines:

- Clarity: Is the code *clear*? Is it easy to read and reason about?
- Function: Does it actually run?
- Correctness: Does the code match the specification? How closely?
- Testing: How thorough is your testing?
- Documentation: How clearly have you explained the code and its API?

Notably, and perhaps unusually, we are looking specifically for engineers who will build things with care and clarity. It's best to finish everything here to a fine polish. Barring that, we will find more of what we seek if you implement half of this exercise with zero defects, than if you implement all of this exercise with bugs. Move steadily and build things.

Should we invite you for an on-site interview, we'll want to discuss your implementation in person. One target for your documentation, then, might be that you'll be able to discuss your code later.

Finally, One Large Hint

The Semantic Versioning specification is rather short for a formal specification, but unless you're already *unusually* familiar with it, it probably describes details that you don't know about. Those details matter!