Project 1

Due: 30 Jan 2009

The first project you’ll work on is a blocks-world planner. Like our running example in class, it involves cubes arranged into stacks, although this world is slightly richer: there are three dimensions (X and Y are the “floor”, and Z is how high in the stack a block is), and each block has an associated colour (one of blue, green, red, or orange).

The input to the system will have two parts. First, a list of facts indicating the dimensions of the room and the current location of every block. (The robot always starts out over location (0,0), and is not grabbing any block.) Then, a list of requirements for a valid final state.

The output of the system will be a simple plan: text according to a particular format that could in theory be handed off to a robot for execution.

1 Spec

1.1 World

The first line of the description file will be of the form

\[(dimx \ dimy \ dimz \ n)\]

These give, respectively, the dimensions along the x, y, and z axes, and the number of blocks in the world. All dimensions are zero-indexed.

The next \(n\) lines of the file will be of the form

\[(name \ colour \ x \ y \ z)\]

and provide the description for a single block. The name is a unique identifier for that block. The only valid colours are blue, green, red, and orange. You don’t need to validate the coordinates, and can assume that no two blocks will be at the same location and no blocks will be floating in midair.
1.2 Goal

Any further lines of the input file will outline requirements of any final state of the system. To be considered a correct solution, all the requirements must be met (that is, there is an implicit “and” around all of them collectively). The asterisks are not part of the grammar but correlate to comments below.

\[
<\text{Reg}> ::= <\text{Simple}>
\]
\[
| \ (\text{not } <\text{Reg}>) *
\]
\[
| \ (\text{and } <\text{Reg}> \ldots) *
\]
\[
| \ (\text{or } <\text{Reg}> \ldots) *
\]
\[
| \ (\text{invariant } <\text{Reg}>) **
\]

\[
<\text{Simple}> ::= (\text{at } <\text{Block}> \ x \ y)
\]
\[
| \ (\text{at } <\text{Block}> \ x \ y \ z)
\]
\[
| \ (\text{on } <\text{Block}> <\text{Block}> \ldots) ***
\]
\[
| \ (\text{over } <\text{Block}> <\text{Block}> \ldots) ***
\]
\[
| \ (\text{touching } <\text{Block}> <\text{Block}>) ****
\]
\[
| \ (\text{samestack } <\text{Block}> <\text{Block}> \ldots) ****
\]

\[
<\text{Block}> ::= (\text{block } \text{name})
\]
\[
| \ (\text{any colour})
\]

NOTE: asterisked items are part of the full spec, but should probably be implemented later than the core subset. Of them, the boolean combinators (one asterisk) should come first, then the invariants (two asterisks), then on/over (three), then touching/samestack (four asterisks).

The boolean combinators have their conventional meaning. Any invariant requirement needs to be true not just at the end of the execution, but at every step along the way (these will probably be not forms, mostly). The semantics of each simple requirement form is as follows:

\textbf{at} This form requires that the specified block is at the specified location. If all three coordinates are given, it says that the block at that location must meet the specification; if only two, it just requires that at least one block in the column meets the specification.

\textbf{on} Indicates that the first block must be immediately atop the second. If there are more than two blocks, indicates a contiguous stack.
over Like on, but with no immediacy requirement (i.e. there can be blocks between them).

touching Always predicated of exactly two blocks, it requires that two blocks meeting the spec must be exactly one apart on exactly one dimension (diagonals don’t count).

samestack Only requires that the listed blocks are in the same column, in any order.

For the two block specification types, (block name) refers to the unique block identified by name. Predictably, (any colour) can match any block of the specified colour (e.g. (any blue) will match any blue block).

1.3 Primitive actions

Just as discussed in class, the robot’s primitive actions will be moveto (which in this case has an x and a y argument), grab, and release. If they are not successful, for whatever reason, they simply do not execute.

The main thing you’ll need to do with these, with respect to your model, is compactly but correctly represent their effects on states, in order to perform your search.

1.4 Interface

The main interface is that your program will read the spec from standard input and write the resulting plan to standard output. If you want to create extra output (for explanation or debugging), send it to standard error!

2 Miscellaneous instructions

Have enough implemented and thought about that we can have a productive conversation in class on Friday. Note that this doesn’t preclude talking to each other outside of class about the problem, too.

Definitely start small. Play with at and the named blocks at first, and work your way up to the more complex requirements more gradually. The asterisked lines in the spec above are my recommendations for order of
attack; implementing the three- and four-asterisk items is not hard on the ‘does it meet the spec’ end, but will make it somewhat harder to effectively shape the search.

Speaking of starting small: your first test cases should be on the order of three blocks in a 3x3 grid, which will probably be harder than you think. Maybe smaller.

3 Things to think about (but not implement)

3.1 Reactive planning: unknown world

Imagine that the robot only could see the top block in each column (if any), so that it would have to revise its plan as soon as it starts moving blocks around and sees what’s underneath. In this version, the information you’d get from the requirements would be incomplete, and you’d have some predicates you could call after each act() to see what was revealed about the world.

3.2 Size and invariants

If the blocks had yet another attribute, size (or weight), you could implement general requirements on what could stack atop other things via the invariant spec. In fact, you could encode the Towers of Hanoi and get your planner to solve it.

3.3 Variables

Syntactically, this would involve an extension of the Block type to also permit

```
| (var foo)
```

for instance, (var X), as in

```
(and (on (var X) (block a)) (touching (var X) (block b)))
```

meaning that whatever block is on top of the block named “a” should be touching block “b”. Variables are required for a lot of the most interesting requirements, but they also make the search space very large.