Concurrency and occam-π

occam-π Exercises (Cylons)

The *Cylons* are a race of robots with a *laser scanner* for vision, *motors* to enable movement and an imperative to explore their environment.

> *"The Cylons were created by man … They evolved … They rebelled … There are many copies … And they have a plan …"*
>
> [Universal Television and Sci-Fi Channel, *"Battlestar Galactica"*, R.D.Moore et al, 2004.]

For this exercise, Cylons live in a walled 2D rectangular world containing free space, in which they can move, and obstacles, through which they cannot move. Only Cylons live in this world – and they are solid and cannot pass through each other.

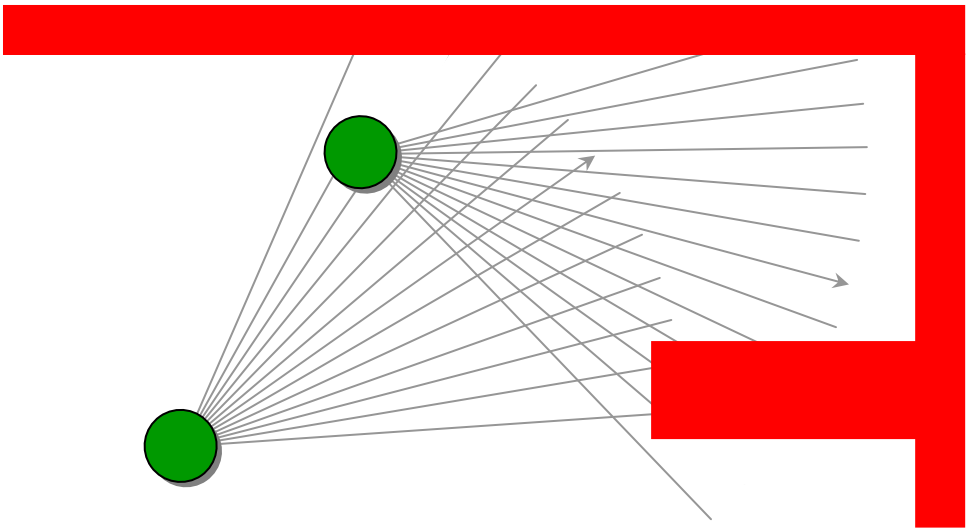

Figure 1: two Cylons approaching a corner.

Figure 1 shows two Cylons, each occupying a circular footprint, approaching obstructions. Laser rays (some of them) are shown emanating from each Cylon. These scan the area ahead through a range of evenly spaced angles. The central ray has an arrow-head (in the figure) to indicate the current direction of motion of the Cylon. The rays are used for range finding and have a maximum length of effectiveness. If a ray strikes an object within that length, a sensor on the Cylon detects the reflection and uses that to determine distance. If no reflection is detected, a distance equal to the maximum effective length of the laser ray is recorded. Either way, an array of distances known to be free of obstacles is obtained – one for each ray angle. Other Cylons are detected by this laser range finder, but the information obtained (only distances) does not distinguish between Cylons, walls and other obstacles. *However, Cylons may be moving.*

Cylon movement is implemented by an undisclosed technology, but operated through two parameters: *linear velocity* (forward speed) and *angular velocity* (rotational speed). No limits are known for these parameters, but Cylons should not push them beyond what their control logic (their *brains*) can safely manage.

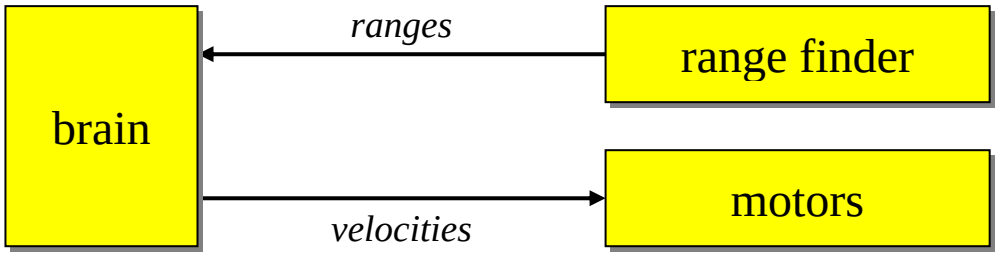

Figure 2: Cylon basic network.

Cylons are built as a network of, at least, the following components (see Figure 2):

- *range finder:* this delivers a stream of arrays of integers – each array holding the set of distances obtained from the latest laser scan;
- *motors:* this receives *linear* and *angular* velocity instructions and makes it so;
- *brain:* this receives the laser scans from the range finder, computes a plan and sends movement instructions to its motors.

This exercise is to program the Cylon brain so that:

- Cylons keep moving as fast as it is safe;
- Cylons do not crash into walls, obstacles or other Cylons;
- They explore as much territory as possible (e.g. they do *not* just run in circles).

However, this exercise is *open-ended* and you are encouraged to devise your own rules – for instance, wall-following behaviours may be interesting. A short report has to be submitted along with your code. Any different rules implemented by your Cylon brain should be explained there. Please communicate with your seminar leader or me (`phw@kent.ac.uk`) if you are unsure whether your rules will be acceptable. Implementation of the basic rules above will be worth around 80% of the marks for this exercise.

The starter file for this exercise is `cylons.occ`. This contains a complete simulated environment for any number of Cylons, together with graphics animation and (limited) user control. The opening *occamdoc* in the file shows the network diagram for the simulation. There are `VAL INT` constants defining the number of robots, number of laser rays per robot, laser scan sweep angle, and sizes for the (raster) graphics window. Leave these alone – certainly to begin with!

The *range finder* and *motors* components for each Cylon are combined into one process, `drone`, in this system. However, each Cylon does have a `brain` process and it is this that has to be (re-)programmed.

In fact, this starter system supports *two* brain processes – `brain.0` and `brain.1`. Interaction at the start of a simulation allows the user to specify which brain the Cylons will use. Also offered is a mixture, with half the Cylons using `brain.0` and the other half `brain.1`. If the mixed option is chosen, `brain.0` Cylons will be coloured red and `brain.1` Cylons coloured green. If a single brain type is chosen, the Cylons will be randomly coloured.

Each brain process has the same header and pattern of synchronization with its environment:

```
PROC brain.x (VAL INT initial.linear.velocity,
              VAL ANGLE initial.angular.velocity,
              VAL INT robot.radius,
              CHAN MOVE move!,
              CHAN LASER laser?)

  INITIAL INT linear.velocity IS initial.linear.velocity:
  INITIAL ANGLE angular.velocity IS initial.angular.velocity:

  ...  other local variables

  SEQ
    ...  initialization code (maybe)
    WHILE TRUE
      RANGES ranges:
      SEQ
        laser ? ranges
        ...  decide new linear.velocity and angular.velocity
        move ! linear.velocity; angular.velocity

:
```

A crucial part of the above behaviour, which must not be changed, is that in each loop cycle a `LASER` message is first input and, then, a `MOVE` instruction is output. These must be the *only* actions on a brain's external channels – otherwise, the system will deadlock.

Note that a brain has no knowledge of absolute position (its coordinates in its world) or absolute bearing. It only knows *how fast* it is moving and turning. This corresponds with real brains *(do you really know where you are and which way you are facing … assuming you don't have a GPS receiver and don't have a fixed external reference … like the stars?).*

The starter file contains a fully programmed `brain.0` and a `brain.1`, which just invokes `brain.0` to do its thinking. Your task is to reprogram `brain.1` (`brain.0` is not too clever).

This process (and the rest of the system) may contain a few aspects of occam-π that have not yet been covered in the course. Those in the rest of the system may be ignored for now. Here, there are possibly three aspects to note: channel *protocols* (`LASER` and `MOVE`), *data types* (`RANGES` and `ANGLE`) and *functions*. For this exercise, all we need to know is:

- `LASER` (defined soon after the opening `VAL INT` constants): a channel carrying this protocol delivers a single item of type `RANGES`.

- `MOVE` (defined just after `LASER`): a channel carrying this protocol delivers two items – an `INT` followed by an `ANGLE`. Channels receiving them must provide two variables (an `INT` followed by an `ANGLE` and separated by a semi-colon). Channels sending them must provide two values (`INT`; `ANGLE`) – as shown above.

- `RANGES` (defined just before `LASER`): this is a *mobile* array of integers. A mobile type can be operated on as if it were the underlying data type – in *OO* terminology, think of it as a *sub-class* (a specialised version of the array *super-class*, inheriting all its capabilities). So, a `RANGES` variable may be indexed (like an ordinary array), `SIZE`d (like an ordinary array) or passed to processes (or functions) expecting an ordinary array, `[]INT` (as is done with the `min` function, used by `brain.0`). Do not worry about the *mobile* aspect of `RANGES` – it allows us to receive arrays of different sizes (not needed here, unless the Cylon laser scanner becomes dynamically adjustable) and gives more efficient transport than *non-mobile* arrays (which is why its here).

- `ANGLE`: this is a specialised form of `INT` (in *OO* terminology, a subclass of `INT`). Anything we can do with an `INT` (e.g. arithmetic) we can do with an `ANGLE`.

  An `ANGLE` divides a full turn into `2^32` units. So, a *right-angle* (a quarter turn) is `2^30` units, *straight ahead* (no turn) is `0`, *straight behind* (a half turn) is `2^31` units, a *degree* is `(2^32)/360` units and a *radian* is `(2^31)/π` units. `ANGLE` arithmetic, like that for `TIMER`s, must allow for integer *wrap-around* (e.g. two half-turns are the same as no turn). So, using the operators `PLUS`, `MINUS` and `TIMES` is required (rather than `+`, `-` and `*`).

  `ANGLE` is defined in the `rastergraphics.module`, imported at the start of the file. Constants are provided for `ANGLE.RADIAN`, `ANGLE.DEGREE`, `ANGLE.RIGHT` … and others. In fact, operators `+`, `-` and `*` are overridden by this module to enforce wrap-around; but `PLUS`, `MINUS` and `TIMES` are preferred (because their semantics are more logical, the code compiled is smaller and their execution costs are lower).

  Look up `ANGLE` in `rastergraphics` in the *"occamdoc documentation for all the occam-pi libraries"* (linked from the course web page).

- `FUNCTION`: occam-π *functions* can take only `VAL` data parameters and return a list (comma separated) of results. No *side-effects* are allowed. They have a weird(ish) syntax. The `brain.0` process uses two of them: `min` and `compute.speed` (each returning a single result). See the *shared-etc* slides for a summary … also presented in the *"occam-pi reference/checklist"* Wiki (linked from the course web page).

Final hints for re-programming `brain.1`:

- find the *maximum* and *minimum* distances in the `RANGES` array;
- if the *minimum* is less than the robot radius, it's bumped – back off quick;
- if the *minimum* is less than twice the robot radius, slow down fast and do something;
- otherwise, head for the direction giving the *maximum* clear distance, basing your speed on how far that is (if there is a tie for *maximum*, go for the one in the nearest straight-on direction) – consider reusing `compute.speed` (which `brain.0` uses);
- sometimes the above rules lead to Cylons getting stuck in corners or in huddles with each other – watch out for this and break out (you'll need new rules here);
- consider using *different* processes for *different* rules (and coordinate them);
- manageable linear velocities lie between `+6` and `-6` (see the `compute.speed` code);
- manageable angular velocities lie between `+10` and `-10` *degrees*.

Please feel free to reject and/or enhance the above. Please be imaginative – we like surprises!