Implementing Reactive Agents using Production Systems

1 Declarative and Procedural Knowledge

We have chosen to represent the knowledge that our simple-minded agent possesses as rules. As we will see as we proceed through the course, there are many other ways of representing knowledge. You might legitimately be wondering why we went to the trouble of using a production system when we could simply have written a short method in Java to achieve the same effect, e.g. something like:

```
public void lightSeeking()
{  while (true)
   {  s0 = leftSensor.read();
      s1 = rightSensor.read();
      if (s0 > s1)
      {  this.turn(LEFT);
      }
      else if (s1 > s0)
      {  this.turn(RIGHT);
      }
      else
      {  this.move();
      }
   }
}
```

The key difference here is between declarative and procedural ways of representing knowledge. The distinction between the two is blurry, but it can be useful.

Declarative knowledge: Knowledge is specified as statements about the world. But how to use that knowledge to achieve some task is not given. Instead, some program that is capable of working with that knowledge must be supplied separately.

In our case, we have the knowledge in the form of rules, and then we must supply the production system interpreter (i.e. the program that searches for matches, resolves conflicts, etc.) separately.

Procedural knowledge: Control information that is needed to carry out some task is embedded into the knowledge.

On the whole, declarative knowledge is to be preferred:

- It can be used for multiple purposes and in multiple ways. You simply supply a different interpreter.
- It can be more easily changed. Insertions, modifications and deletions are easy. This means that machine learning becomes more feasible.

Procedural knowledge has the advantage that, being specialised to one task, it can often accomplish that task more efficiently than would a declarative approach.

Keep in mind that there is no clear-cut distinction. For example, some people would regard our rules as being at least partially procedural because they specify actions.

We can illustrate the flexibility of rules by considering how we might encode default actions and goals using a production system. In both cases, note how little needs to be changed.

2 Default Actions in Production Systems

Suppose you have a production system whose conflict resolution strategy is to always choose the first-matching rule. When devising the rules for your production system, it is conventional for the last rule to have as its condition simply **true** :

$$\textbf{if}\quad \textbf{true}\quad \textbf{then}\quad a_{\text{default}}$$

So if no other rule (higher in the list) is used, this rule will be used as a last resort. The action in this rule, a_{default}, will be some suitable default action to be executed when nothing else is suitable. This may prevent your agent from being paralysed by inaction.

Question *Here again are the rules for the agent from the previous lecture:*

$$\begin{array}{llll} \textbf{if} & p_0 & \textbf{then} & \text{TURN\_LEFT} \\ \textbf{if} & p_1 & \textbf{then} & \text{TURN\_RIGHT} \\ \textbf{if} & \neg p_0 \wedge \neg p_1 & \textbf{then} & \text{MOVE} \end{array}$$

Under what circumstances did he ‘freeze’?

How would you rewrite him so that he could get himself out of these difficulties?

3 Agents with Goals

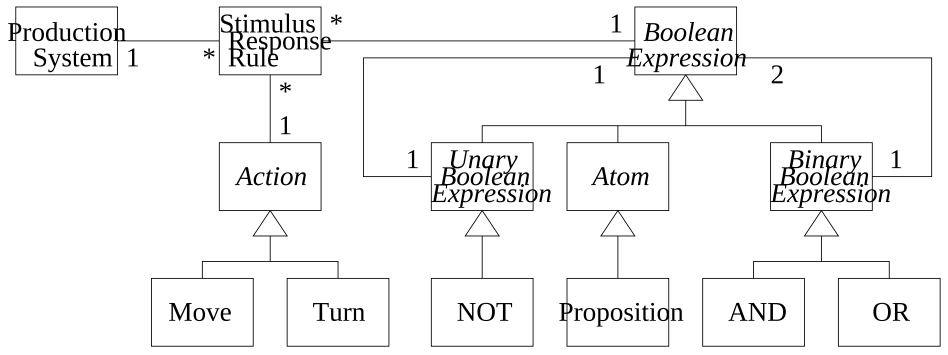
At the moment, our agents are always ‘on the go’. At every moment of time, they sense, plan and act. Suppose instead you want your agent to achieve some goal and then stop.

Question: *How would you write your production system to achieve this?*

4 Java Implementation of a Production System

As you’ve seen, I’ve written a system in which agents live in grid-like graphics worlds, where they encounter bricks and other agents. We’ll study enough of this code to allow you to experiment with your own agents. We look most closely at how the rules in a production system are implemented because this information will be used in the next lecture.

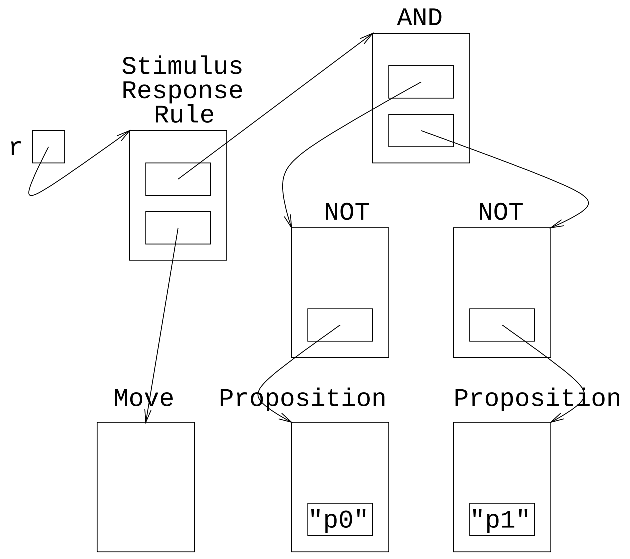
A class diagram for part of the system:



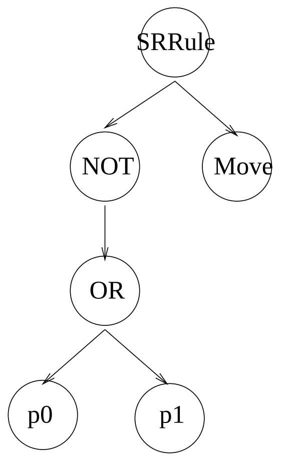
Here's code for creating a stimulus-response rule that corresponds to **if** $\neg p_0 \wedge \neg p_1$ **then** MOVE:

```
StimulusResponseRule r =
   new StimulusResponseRule(
      new AND(new NOT(new Proposition("p0")),
              new NOT(new Proposition("p1"))),
      new Move());
```

In memory, variable `r` will now contain a reference to a tree-like object:

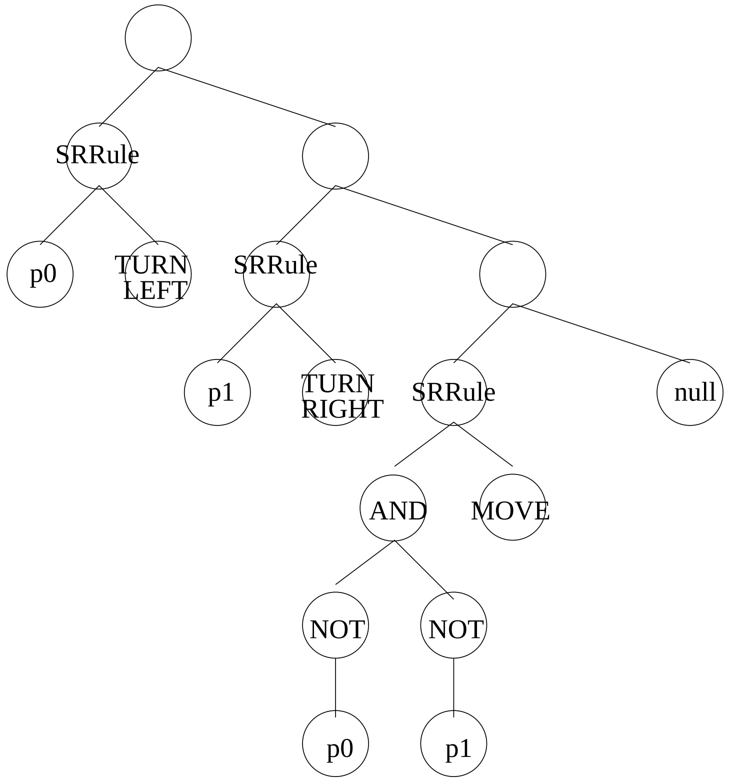


Question: *What rule is being represented by the following tree? (The picture suppresses some of the details that cluttered up the previous one).*

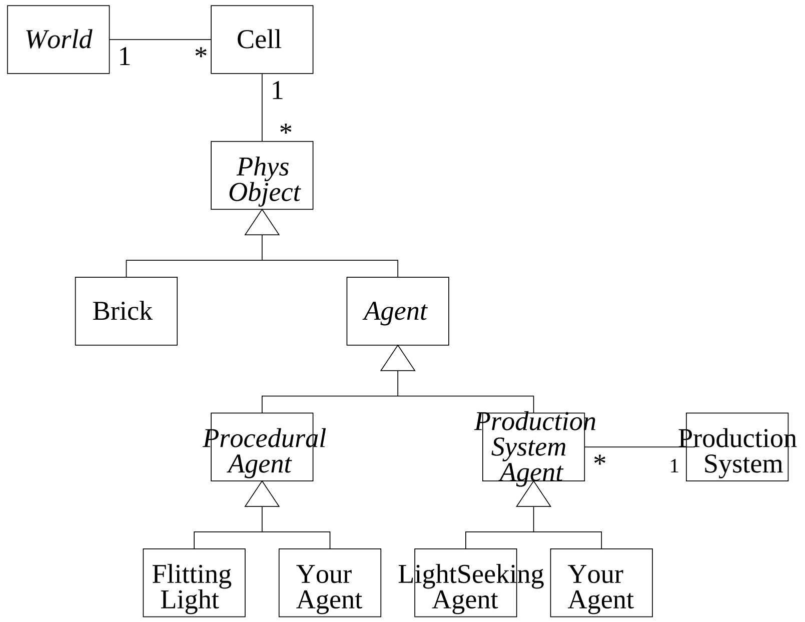


How would you create it using the Java constructors from above?

A production system is a list of rules. But, in memory, lists are also stored in node-and-pointer tree-like structures (assuming they're linked-lists rather than arrays). So a production system might be stored in a tree as follows:



There is no need to understand the rest of the system, which is concerned with organising the simulation in the grid-like world. For those of you who are interested, its class diagram follows. (Even this diagram is a simplification: I have excluded the numerous GUI and listener classes!)



Questions

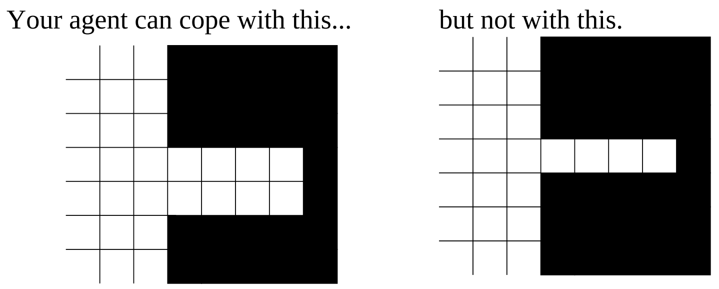
1. Consider the wall-following agent demonstrated in the lecture. Design its production system.

   Suppose the agent has 8 touch sensors and these have been used to give 8 propositions $\langle p_0, \ldots, p_7 \rangle$. Variable `p0` contains `true` if there's an object directly in front of the agent; `p1` is true if there's an object in front but to the right of the agent; `p2` is true if there's an object directly to the left of the agent; and so on, as shown by this diagram:

p7	p0	p1
p6	◯	p2
p5	p4	p3

   The actions are `Move` and `Turn`. `Move` moves the agent one pace forward in the direction that it is facing. For `Turn`, we supply two arguments. The first argument is either `LEFT` or `RIGHT`. The second argument is an integer that indicates how many $45°$ rotations make up the turn. For example, `Turn(RIGHT, 2)` is a turn action that turns this agent 90 degrees to the right (e.g. if it was facing North, it is now facing East).

2. Our agent is unable to handle 'tight dead-end corridors', i.e. a dead-end path between two walls that is less than two cells wide:



Indeed, no purely reactive agent can handle these. Why not?

5 Appendix: Propositional Logic

For those who need it, here is a quick recap on some immediately relevant parts of Propositional Logic.

In Propositional Logic, we're interested in analysing *statements* about the world, and these statements will be either **true** or **false** . (This is a major simplifying assumption made in Propositional Logic: it excludes the possibility of statements that are, e.g., only partly true. To handle partly-true statements, you have to abandon Propositional Logic and turn to one of the many other varieties of logic, e.g. Fuzzy Logic — covered later in this course.)

For conciseness, we do not write statements such as 'There is an obstacle in front of the agent' or 'The amount of light entering the left eye exceeds the amount entering the right eye'. Instead, we give these statements short names such as $p, p_0, p_1, \ldots, q, q_0, q_1, \ldots$ We refer to these as *propositional symbols*.

We can construct more complex statements from simpler statements by combining propositional symbols into larger expressions using operators that we call *connectives*. We can work out whether the larger expression is **true** or **false** from the truth-values of the propositions contained within that larger expression.

We will look at the five main connectives.

Negation Given any expression W, we can construct another expression, the *negation* of W, which we write

$$\neg W$$

The truth/falsity of $\neg W$ can be computed from the truth/falsity of W as follows:

W	$\neg W$
true	**false**
false	**true**

This corresponds loosely to uses of the word 'not' in English.

Conjunction Any two expressions, W_1 and W_2, can be combined to form a compound expression called the *conjunction* of W_1 and W_2, which is written

$$W_1 \wedge W_2$$

W_1 and W_2 are called the *conjuncts*.

The truth/falsity of $W_1 \wedge W_2$ can be computed from the truth/falsity of W_1 and W_2:

W_1	W_2	$W_1 \wedge W_2$
true	**true**	**true**
true	**false**	**false**
false	**true**	**false**
false	**false**	**false**

This corresponds loosely to uses of the word 'and' in English.

Disjunction Any two expressions, W_1 and W_2, can be combined to form a compound expression called the *disjunction* of W_1 and W_2, which is written

$$W_1 \vee W_2$$

W_1 and W_2 are called the *disjuncts*.

W_1	W_2	$W_1 \vee W_2$
true	**true**	**true**
true	**false**	**true**
false	**true**	**true**
false	**false**	**false**

This corresponds loosely to uses of the word 'or' in English. However, it is more precisely 'inclusive-or' since it covers the idea of 'W_1 or W_2 or both'.

There is another connective, referred to as 'exclusive-or', sometimes written as $W_1 \oplus W_2$, whose truth/falsity is computed as follows:

W_1	W_2	$W_1 \oplus W_2$
true	**true**	**false**
true	**false**	**true**
false	**true**	**true**
false	**false**	**false**

This covers the idea of 'W_1 or W_2 but not both'.

Conditional Any two expressions, W_1 and W_2, can be combined to form a compound expression called a *conditional*, which is written

$$W_1 \Rightarrow W_2$$

W_1 is called the *antecedent* and W_2 is called the *consequent*.

W_1	W_2	$W_1 \Rightarrow W_2$
true	**true**	**true**
true	**false**	**false**
false	**true**	**true**
false	**false**	**true**

It is often said that this corresponds loosely to uses of 'if . . . then' in English. However, the correspondence is not strong and can be misleading.

Production rules (**if** W **then** a_i) and conditionals ($W_1 \Rightarrow W_2$) are completely different beasts. A production rule associates a condition, W (an expression of Propositional Logic), with an action, a_i. A conditional connects two expressions of Propositional Logic, W_1 and W_2, to make a larger statement about the world.

Bi conditional Any two expressions, W_1 and W_2, can be combined to form a compound expression called a *biconditional*, which is written

$$W_1 \Leftrightarrow W_2$$

W_1	W_2	$W_1 \Leftrightarrow W_2$
true	**true**	**true**
true	**false**	**false**
false	**true**	**false**
false	**false**	**true**

This corresponds loosely to uses of 'if and only if' in English. However, again the correspondence is not strong and can be misleading.

Now, if your agent knows that p_1 is **true** , p_2 is **false** and p_3 is **true** , it can use what we learned above to compute the truth-value of more complex expressions such as $p_1 \wedge (p_2 \Rightarrow p_3)$. (It comes out **true** .)

Although an agent will probably never want to do this, there are some occasions when logicians want to compute the truth value of an expression for all possible assignments of **true** /**false** to its propositional symbols. When we do this, we show the results as a *truth-table*. Here's an example:

p_1	p_2	p_3	$p_1 \quad \wedge \quad (p_2 \Rightarrow p_3)$
true	**true**	**true**	**true**
true	**true**	**false**	**false**
true	**false**	**true**	**true**
true	**false**	**false**	**true**
false	**true**	**true**	**false**
false	**true**	**false**	**false**
false	**false**	**true**	**false**
false	**false**	**false**	**false**

If there are n distinct propositional symbols, then there are 2^n different rows in the truth-table. (Here $n = 3$ so there are $2^3 = 8$ rows.)

Inspection of a truth-table allows us to say things about an expression.

- An expression is *satisfiable* if there is *at least one row* in the truth-table where the expression comes out **true** .
- An expression is *unsatisfiable* if there is *no row* where the expression comes out **true** .
- An expression is *valid* if *every row* comes out **true** .
- Two expressions, W_1 and W_2, are *logically equivalent*, written $W_1 \equiv W_2$, if they come out the *same in each row*.

There is another way of showing that two expressions are logically equivalent, and that is algebraically. For this, you make use of a number of laws. Laws are familiar from arithmetic. For example, you know that $3 + 4 = 4 + 3$ and, in general, that $W_1 + W_2$ always gives the same answer as $W_2 + W_1$.

Here are the laws of Propositional Logic.

For any expressions W, W_1, W_2 and $W_3, \ldots$

Commutativity of $\wedge$ and $\vee$

$$W_1 \wedge W_2 \equiv W_2 \wedge W_1$$
$$W_1 \vee W_2 \equiv W_2 \vee W_1$$

Associativity of $\wedge$ and $\vee$

$$(W_1 \wedge W_2) \wedge W_3 \equiv W_1 \wedge (W_2 \wedge W_3)$$
$$(W_1 \vee W_2) \vee W_3 \equiv W_1 \vee (W_2 \vee W_3)$$

Distributivity of $\wedge$ over $\vee$, and of $\vee$ over $\wedge$

$$W_1 \wedge (W_2 \vee W_3) \equiv (W_1 \wedge W_2) \vee (W_1 \wedge W_3)$$
$$W_1 \vee (W_2 \wedge W_3) \equiv (W_1 \vee W_2) \wedge (W_1 \vee W_3)$$

De Morgan's laws

$$\neg(W_1 \wedge W_2) \equiv \neg W_1 \vee \neg W_2$$
$$\neg(W_1 \vee W_2) \equiv \neg W_1 \wedge \neg W_2$$

Idempotence of $\wedge$ and $\vee$

$$W \wedge W \equiv W$$
$$W \vee W \equiv W$$

Identities

$$\textbf{true} \wedge W \equiv W$$
$$\textbf{false} \wedge W \equiv \textbf{false}$$
$$\textbf{true} \vee W \equiv \textbf{true}$$
$$\textbf{false} \vee W \equiv W$$

Involution

$$\neg\neg W \equiv W$$

Complement laws

$$W \wedge \neg W \equiv \textbf{false}$$
$$W \vee \neg W \equiv \textbf{true}$$
$$\neg\textbf{true} \equiv \textbf{false}$$
$$\neg\textbf{false} \equiv \textbf{true}$$

Definition of biconditional

$$W_1 \Leftrightarrow W_2 \equiv (W_1 \Rightarrow W_2) \wedge (W_2 \Rightarrow W_1)$$

Definition of conditional

$$W_1 \Rightarrow W_2 \equiv \neg W_1 \vee W_2$$

So, we can use these laws to show, for example that $p \vee q \equiv \neg(\neg p \wedge \neg q)$ Typically, we pick one of the expressions and use the laws to transform it into the other expression. Here I have picked the right-hand side expression and transformed it, in two steps, into the left-hand side expression:

$$\begin{array}{rcll} \neg(\neg p \wedge \neg q) & \equiv & \neg\neg(p \vee q) & \text{(by De Morgan's law)} \\ & \equiv & p \vee q & \text{(by Involution)} \end{array}$$

(If only they were all as easy as that. . . Now tackle the exercises!)

Appendix Exercises

1. Draw truth tables for the following expressions *and hence* state whether each is valid, satisfiable and/or unsatisfiable.
   (a) $(p \Rightarrow q) \Rightarrow (p \wedge q)$
   (b) $(p \Rightarrow q) \vee \neg(p \Leftrightarrow \neg q)$
   (c) $p \wedge (q \Leftrightarrow \neg q)$
   (d) $p \vee \neg(p \wedge q)$
   (e) $p \vee \neg(p \wedge p)$
   (f) $\neg(\neg(p \wedge q) \wedge \neg(p \vee q))$
   (g) $((p \wedge \neg q) \vee (\neg p \wedge q)) \Rightarrow r$
2. Here are some valid expressions:
   $$p \Rightarrow p$$
   $$p \vee \neg p$$
   $$\neg(p \wedge \neg p)$$
   $$((p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_3)) \Rightarrow (p_1 \Rightarrow p_3)$$
   Here's an unsatisfiable expression:
   $$p \wedge \neg p$$
   (You can optionally confirm that these expressions are valid/unsatisfiable by drawing truth-tables. This would be good practice for you.)
   (a) If an expression is valid, then its negation is unsatisfiable. Why?
   (b) If an expression is unsatisfiable, then its negation is valid. Why?
   (c) *Hence*, answer the following *without drawing truth-tables*.
      i. Is $\neg(p \vee \neg p)$ valid? Is it satisfiable or unsatisfiable?
      ii. Is $\neg(p \Rightarrow p)$ valid? Is it satisfiable or unsatisfiable?
3. Use the laws of the Propositional Logic to show that the following equivalences hold.
   (a) $\neg(W_1 \Rightarrow W_2) \equiv W_1 \wedge \neg W_2$
   (b) $W_1 \Rightarrow W_2 \equiv \neg W_2 \Rightarrow \neg W_1$
   (c) $(W_1 \wedge W_2) \Rightarrow W_3 \equiv W_1 \Rightarrow (W_2 \Rightarrow W_3)$