Contents

AI and PopBeast

Jocelyn Paine

January 12, 1998

This is the second handout for the AI practical. I'm going to talk about a small classical-AI program — Popbeast — which "lives" in the computer-generated environment Eden. This program carries out actions in response to commands typed in English. As it does so, it displays the representations it is building up, so you can watch its "mind" in action. This handout will introduce Eden itself, and also some ideas from AI. Unfortunately, PopBeast is not robust enough for you to experiment with it. It's still a useful example, though.

So what is PopBeast? If we had unlimited resources, I could have considered a robot, complete with TV camera and other sensors. However, robot hardware is not cheap, and given the limited capabilities of today's robots, I would also have problems in designing a suitably limited environment. Until recently, AI has found it much easier to make programs that indulge in "rational thought" — that play chess, translate languages, or prove theorems — than to make programs that, by perception and action, can link their thoughts to the real world. Most industrial robots, if they use vision at all, only do so in a very limited form; some robots are still guided by touch, and find it impossible to cope if the parts they're assembling aren't exactly where they should be.

Moreover, I wanted, in this handout, to illustrate some concepts from classical AI — particularly the notion of representation, as well as examples of the type of "mental modules" you would find in a classical AI program. Many of the more recent robots do not use these techniques, and so wouldn't be suitable examples anyway.

1 Eden and microworlds

I have therefore settled on a program which acts not within the real world, but within a computer, in a "microworld" called *Eden*. Up until the Eighties, much of AI was carried out in such microworlds, working on the principle that if you start small, and simplify a problem to its utmost by working in a simulated environment stripped of all inessentials, you can scale up later to the real world. Thus a lot of research in planning has been done in the *blocks world*, which consists of a few blocks whose only attributes are colour and size, stacked on a table. In general, such environments are known as *microworlds*. One of the most famous early AI programs, *Shrdlu*, was based in a microworld: see, e.g. the account in *AI and Natural Man* by Boden.

2 Classical AI

There's a common approach to AI that pervades most of the methods developed in such settings. This is often called the *classical* approach. As described in my introductory notes, it's associated with the assumption that any intelligent agent (robot, etc) needs a memory in which it stores information about the current state of the world. As the world changes, the robot's perceptions will change with it. The robot must decode these perceptions and update the information in its memory. In effect, this information forms the robot's model of the world. The robot also has certain goals. By combining these with the information in memory, it can make plans. These consist of sequences of primitive movements, which are fed to a "motor control" unit and obeyed. The classical approach also tends to break down agents by *vertical decomposition*, concentrating on one task and studying that in isolation from others. EWven though PopBeast can do several — analyse sentences, build world-models, plan — it is not a complete agent, as it has no way to formulate its own goals. It is not *autonomous*.

This will become clearer when you see the PopBeast examples. Although the classical approach has come in for a lot of criticism during the last ten years, it was the dominant paradigm for the previous twenty-five, and it has left a large legacy of terminology. So it's definitely worth demonstrating.

3 Eden

PopBeast lives in a microworld called Eden, designed to provide an environment which is reasonably challenging whilst being easy to compute. Eden was designed by Simon Perkins for the AI Society, as a setting for members to write and test AI programs; since then, I've updated it in various ways. This was in about 1991, when the terminals available to undergraduates were almost all text-only VT100-compatibles. Given these limitations, he did a good job of providing a challenging, yet conceptually simple, world within the constraints of Oxford's computers.

Actually, there were some PCs around already, but the mainframe of the time (a VAX) couldn't easily generate graphics on them. With a fair amount of careful programming this could have been arranged, but the Department was reluctant to fund improvements to the course by paying for programming graphical user interfaces and similar. They still are, but the advent of the Java programming language means such work may now be a practical spare-time proposition. This is why the parts of the course where you'll be running your own programs use Eden II, not Eden.

Given that Eden had to be displayed on text-only terminals, what does it look like? Strictly speaking, you can have many different worlds, but I'll refer to them all as "Eden". Figure 1 shows one possibility, with an explanation of the symbols underneath.

Eden is two-dimensional, and divided into a rectangular grid of squares. Each square can hold one object, and may also be able to accomodate PopBeast. However, whether this is possible depends on the object's "size". Boulders, rocks, and doors are big enough to block a PopBeast from moving into the square they occupy; food, hammers, and keys are smaller, and can be in the same square as a PopBeast.

How does a PopBeast interact with its environment? A PopBeast has a "retina" and a "sense of smell". Both these are automatically updated by the simulator so that they always reflect the current state of the world. The retina is a 7×5 array representing PopBeast's immediate environment. The sense of smell gives the general direction of the food, as one of `forward`, `back`, `right`, `left`. In the world shown above, PopBeast is facing east (right): it smells the food as being `forward`, and it sees its retina as shown in Figure 2. PopBeast's position is represented by a `B`.

These are severe sensory limitations, which represent a compromise. PopBeast needs *some* senses: it is difficult to build intelligent behaviour into a being with no sensory apparatus whatsoever. On the other hand, the senses must not provide information that's too precise, because one of Eden's objectives is to let people experiment with the problems of perceptual decoding. Given the size of the retina,

```
**********************************************
* B        @ *            *    T             *
*      QQQQQQ*            *                  *
*      QQQQQQ*        ***********            *
** ** @QQQQkQ*        #         *            *
*      QQQQQQ*        *********Q             *
*      QQQQQQ*                *Q             *
*          @ *                *Q             *
*            *                *              *
*        T   *                *              *
*            *********        **************@*
*            *                             *k*
*            @                      ********@*
*            *                      *+#      *
*            *                      **********
*            *                       *       *
**********************************************
```

```
B = PopBeast    - aka Bug.
+ = food        - eat this to gain energy.
* = boulder     - impossible to move through.
@ = rock        - can be smashed with hammer (T).
# = door        - can be opened with key (k).
Q = quicksand   - drowns you unless you immediately move off it.
T = hammer      - smashes a rock.
k = key         - opens a door.
```

Figure 1 Eden.

```
**  * QQ
 *
 *
 *
 *
 *B
 *
```

Figure 2 PopBeast's retina.

if PopBeast is to do any more than react to its immediate surroundings, it somehow needs to move around the world and combine successive retinal images into a consistent and correct *world model*.

As mentioned in my introduction, it is an increasing trend that not everyone agrees on the need for a world-model; some people believe that trying to construct and maintain one leads to inefficient and non-adaptable agents. The contrary view is put by R A Brooks in *Challenges for Complete Creature Architectures*, from *From Animals to Animats*, edited by Meyer and Wilson (MIT 1991; PSY DB:M 057). For a later review, see *Being There* by Andy Clark (MIT 1997; PSY not yet catalogued).

PopBeast also faces the problem of integrating sight and smell. This is harder than it might seem, because the senses are so different. If PopBeast can see the food at all, then it knows exactly where it is. However, its sense of smell only gives a general direction. So the world model must be able to accomodate imprecise data: "hints" about the state of the world. The problem of merging information from different sources is an important one in robotics, and is known as *sensory fusion*.

Having got this sensory information, how does PopBeast use it? The simulator works in a continuous cycle, rather like the turns in a game. At the start of each turn, it updates PopBeast's senses so they're consistent with the current state of the world. It then activates PopBeast's "brain". This is a chunk of computer program which takes the sense data, "thinks", and eventually comes up with a "motor command". The simulator then takes this action, modifies the world accordingly, and then starts the next turn. Warning: these "motor commands" are single symbols, and have very little in common with motor commands in real animals.

At the end of each turn, PopBeast can perform one of eight actions. Four are moves: it can move one step forward or one step back, or turn 90 degrees left or right. It can't move diagonally, or more than one square at a time. Every time PopBeast moves, it loses one unit of "energy": starting with 500 units, if the energy gets down to zero, PopBeast will "die". This, plus the restricted range of movements, make it vital to plan efficient paths when travelling. You also have to avoid obstacles. You can only move forward or back on a quicksand; if you turn, you'll fall in. You can't move through a boulder, rock or door. Boulders represent fixed obstacles; rocks and doors are not so stubborn, because you can smash or open them as described below.

Of the other four actions, two are to do with carrying objects. PopBeast can pick up an object (provided that its "hand" is empty), or put one down (if there's room for it in the square). It can't hold more than one object at a time, however. Again, this restriction enforces efficient planning. Some objects — boulders, rocks, doors — can't be picked up; food, keys and hammers can.

PopBeast can also "use" the object it's carrying. What effect this has depends on what the object is. Using food is equivalent to eating it. Using a key will open a door, if PopBeast is standing next to the door. Using a hammer will smash a rock, if PopBeast is standing next to the rock. Keys and doors are not quite the same as hammers and rocks; if you use a key, it disappears, but if you use a hammer, it remains for you to use again. Using the other objects has no effect.

You might like to think about what the PopBeast in Figure 1 has to do in order to get at the food. Note that it has to open the door in the top middle of the world, but it needs a key to do this, and the only accessible key is hidden in a nest of quicksands.

4 Mind design

How might one design a PopBeast which can survive in such worlds? The answer depends on how much information one is willing to program it with. The PopBeast

I describe below already "knows" the basic properties of keys, doors and other objects when it starts life. However, a truly artificial intelligence ought to be able to learn these for itself, just by wandering around the world, playing with things, and noting regularities. At first sight, it might seem easy to build something which does this — which notices that when you grab a rock it doesn't move, but when you grab a key it does, and therefore keys can be picked up. Indeed, in Eden, building such a program would not be too difficult. Objects always behave in the same way; there are very few objects, and very few actions one can perform; and perceptual data (vision, anyway) is always complete.

In the real world, objects have many more properties and interact in many more ways. Building an AI program that can decide *what* properties and interactions to take note of is not trivial — look up *Bongard problems* in Hofstadter's *Gödel, Escher, Bach* (Penguin 1981: PSY KH:H 67) if you are not convinced.

Moreover, real world objects differ in all sorts of inessential ways (consider the difference between two poodles); there are many more things you can do with them; and no two situations are ever alike. We are a long way from solving the problem of *machine learning*, and I have circumvented it by pre-programming PopBeast with knowledge of its world's basic physics. This is a big cop-out, since the ability to learn should probably be central to every intelligence, with everything else depending from it.

5 PopBeast and information representations

As I hinted earlier, I'm using PopBeast to demonstrate representations. An important idea in AI is that the same information can be represented in many different ways. There's an analogy with chemistry here. How does diamond differ from graphite? Graphite is soft enough for pencil leads; it's opaque, it's a good conductor of electricity, and (as charcoal) it makes an excellent fuel. Diamond is the hardest natural substance known; it's transparent, it doesn't conduct electricity, and not even the most desparate barbecue fanatic would throw his wife's wedding ring into the coals. Yet both are made from the same atoms.

The difference is form. In graphite, the atoms are arranged in layers that slip easily over one another, allowing the free passage of electrons, and coming off in streaks onto paper. In diamond, the atoms form a tetrahedral structure; it has no "give", and you must work very hard to detach an atom.

This is obvious to any chemist. But until 1860 or so, the difference was inexplicable. Although Dalton had put the idea of atoms combining in definite ratios, the notion of shape and arrangement of atoms was missing; no-one seems to have considered it until Van't Hoff proposed that carbon atoms have a definite shape, with bonds pointing outwards as from the centre of a tetrahedron.

Information has form as well. Programmers know that the same piece of information can be represented in many different ways: numbers in base 10, base 2, or as Roman numerals; dates as day/month/year or as days since January 1st 1890; images as arrays of intensity points or as a set of classified objects; geometric facts as drawings on graphs or co-ordinate equations . . . .

These choices are not arbitrary. The way something is represented on a particular information-processing system crucially affects the types of operation we can perform naturally and quickly (Figure 3). For instance, if we represent dates as days since a base date, it's easy to find the time between two dates, but hard to tell which month any date falls into. With the day/month/year representation, the converse is true.

So representation should be a crucial idea in psychology. But until the advent of computational models it has been largely neglected. What made us realise its

```
Picture omitted. It was a cartoon, taken
from a book on software design by Cliff Jones, of an ancient Egyptian
scribe drawing on a large wall-board whilst complaining to somebody who
is dictating near by. He is busy drawing line after line of tiny
man-like hieroglyphs, following a few other symbols. The caption is
``are you sure there isn't a better way of writing `The Pharoah had
10,000 soldiers?' ''.
```

Figure 3 Some representations are better than others.

```
**********************
*                    *
*            * * *   *
*            * + *   *
*            *   *   *
*                    *
**************   *   *
*      *     *  *    *
*   k  *     @     * *
*      *     *       *
*  B   #     *       *
*    * *  T  *       *
*      *     *       *
**********************
```

Figure 4 A simpler Eden.

importance was the technical difficulty of programming computers. Pages 24–33 of *Artificial Intelligence* by Winston (2nd edition) give an excellent example of different representations in an early analogy-solving program.

6 How PopBeast represents its world

Let's start then with the simple Eden of Figure 4.

Here, PopBeast is in a room which contains a key `k` and a door `#`. Suppose that I have told PopBeast to open the door. How does it work out what to do? To answer this question, you have to know how the command "open the door" is presented, and I'll get on to this later. I will start though by talking about how PopBeast represents the world — how it remembers where it is, and what objects are around it.

It does this in two main ways. The first is by maintaining a kind of internal image. In programming terms (if you know any — don't worry if not), this image is a two-dimensional array of characters which holds a copy of the picture above. Non-programmers can think of it as just a two-dimensional grid on which is drawn a map of the world. This is an *analogue* representation. PopBeast builds up the map by wandering around the world and superimposing successive retinal images; this is fairly straightforward, and I shan't say any more about it.

The second method is based on logic, and describes the world as a set of *propositions*. If you look at Figure 5 and compare it with the Eden in Figure 4, you should be able to work out what's going on.

```
square(me, [4,4])
square(key(1), [5,6])
square(door(2), [8,4])
square(hammer(3), [11,3])
square(rock(4), [14,6])
square(food(5), [16,11])

in(me, 49)
```

```
in(key(1), 49)
blocked(49, 50, door(2))
in(hammer(3), 50)
blocked(50, 51, rock(4))
in(food(5), 51)
joins(49, 50)
joins(50, 51)
joins(51, 49)

dir([1,0])
```

Figure 5 Eden in propositions.

For those who don't know logic, a brief explanation of terminology is in order. A proposition is a statement about an object, or about a relation between objects. It is usually written as a name (the *predicate*) followed by some things in brackets. These things are the objects that the proposition is talking about, and are called its *arguments*. If there are more than one argument, they are separated by commas. Note that some authors use variants of this notation: Charniak and McDermott, for instance, use bracketed lists

```
( predicate argument-1 argument-2 )
```

The arguments can be numbers, or simple names like `me`, or groups of things like `[1,0]`, or compound "structures" like `key(1)`. How these are interpreted depends entirely on the program using them: they have no intrinsic meaning. In PopBeast's world model, most objects are represented as their type — `key(_)` or `door(_)` together with a unique number to distinguish one individual from another. An exception to this rule is `me`, for obvious reasons. Co-ordinates are represented as pairs `[X,Y]`, and regions are given arbitrary numbers in the range 49–57 (there are technical reasons for this, which I shall ignore).

Each predicate expresses some fact about an object or set of objects. In this figure, the first group of propositions is all of the form `square( _, [_,_] )`. These describe the contents of squares. The first argument denotes the object within a square, and the second argument, specifying the position of the square, is a pair of (X,Y) values. These are in PopBeast's internal co-ordinate system, for which (0,0) is at the bottom left.

The second group describe the way the world is divided into regions, and how they're connected. I have made PopBeast describe the world in terms of regions because of their importance in everyday life: they enable you to divide the world into pieces which can be reasoned about in isolation, in that influences are contained, i.e. an event happening in one region has very little effect on events in another.

In this group of propositions, the numbers 49,50,51 refer to the three different regions. The `joins` propositions tell you which regions are adjacent; the `blocked` propositions tell you whether you can pass between them, and if not, why not; the `in` propositions tell you which objects lie within a region.

The third group describes PopBeast's knowledge of the direction it is facing in, by specifying how it would move were it to go forward. At the moment, it would go one place right (X) and none north (Y), so the direction is `[1,0]`.

It goes without saying that I am not asking you to learn the details of these notations, merely to appreciate the kind of information they carry. The propositional representation is very close to how the *Prolog* programming language works, and you will see it again in the AI practical.

So far, we've seen three different ways of describing where things are. We can use an analogue internal map, or we can use propositions. Within the propositions, we can either give exact co-ordinates, or we can describe the world in terms of regions and their contents. Why do we need all these? The answer is that different representations are good for different jobs. The analogue map is useful when you know the locations of two points, and you want to plot a detailed path between them. It is also easy to build up from retinal images, and serves as a starting point for the propositional representations. The propositions about squares and object co-ordinates are useful when deciding *which* points we want to plot a path between. The propositions about regions are useful in formulating high-level plans about moving from region to region. If we know that region 1 contains a hammer, and region 2 contains a rock, then to smash the rock in region 2, we have to go into region 1. At this level of detail, the exact co-ordinates of the hammer and rock don't matter, so we can speed up computation by ignoring them. At a lower level, we will eventually need to take them into account: then we descend through the other two representations.

7 How PopBeast represents meaning

Like Shrdlu, PopBeast can obey simple commands. It can also answer questions. How does it represent their meaning? As with the non-analogue part of its world model, it uses propositions. Thus "There is a key in a room" would become

```
in( K, R ), key(K), room(R).
```

Literally translated, this means "K is in R, and K is a key, and R is a room": by "room", PopBeast understands what I called a "region" earlier. The comma is the way that Prolog displays a logical "and". The K and R are what are called *logical variables*. In the world-model propositions, the arguments always named particular objects. Here by contrast, they could refer to *any* objects for which the proposition is true: that is, to any key K and room R such that K is in R. This is reasonable, since the sentence did not specify a particular room or key. Logicians may be puzzled by the lack of a quantifier for the variables K and R. As written in standard logic, this would have a $\exists K \exists R$ quantifier in front of it, but in Prolog, this can be taken for granted. Non-logicians can ignore the last remark.

Verbs are also represented by propositions. Thus the sentence "PopBeast opens a door" would become

```
opens( me, D ), door(D).
```

Literally translated from the logic, this means "I (PopBeast) open D, and D is a door".

There is actually slightly more to the representation than this. With each assertion, PopBeast stores a special marker to indicate that it *is* an assertion rather than a command. The reason for this is that the logical translation alone is insufficient to distinguish the two. The formula

```
opens( me, D ), door(D).
```

could, as above, say something about the current state of the world. However, it could also describe a state of the world which we want PopBeast to make true, but which isn't true now — i.e. act in such a way that you are opening a door. PopBeast can tell from the grammatical structure of a sentence whether it is meant to be a command or an assertion. To summarise, the two meanings are represented as

Sentence	Meaning
PopBeast opens a door	`opens(me,D),door(D) / assertion`
Open a door	`opens(me,D),door(D) / command`

where the `assertion` and `command` markers are not part of the proposition.

You may have noticed that above I have talked about the sentence "Open a door", whereas earlier I said that I could give PopBeast the command "Open the door". What is the difference in meaning? It is often the case in English, though not always, that "the" refers back to some previously mentioned item, or points at an item which is special in some way to the listener or speaker. To deal with this properly, PopBeast would have to determine *which* item this is, possibly by examining the context built up by previous dialogue. I have finessed this by ignoring it, and fixing PopBeast so that "the" and "a" are treated alike.

8 How PopBeast creates plans and actions

As before, let's suppose we have told PopBeast to open the door. Suppose also that it is working on the first set (`square`) of propositions in Figure 5 — to keep these notes short, I'll ignore anything it does with the `in`, `joins` and `blocked` ones.

When I tried this out, with PopBeast as shown in Figure 4, it generated the following sequence of actions:

```
[left, forward, right, forward, left, forward,
 grab,
 right, forward, forward, right, forward, forward, left, forward,
 use
]
```

To check these, you have to know that it was facing East when it started. The notation I have used here is how Prolog displays what's called a *list*. This is a kind of structure that can hold an arbitrary number of data items. Programming languages that provide lists also allow you to join lists together and insert elements anywhere inside them. Hence lists are very good for storing variable-length data like sentences, musical scores, and action sequences. If you know what arrays are, think about the difference. Arrays are fixed length and cannot be extended.

In the computer, lists themselves are implemented as blocks of bits connected in a certain way. However, you need not worry about that because Prolog takes care of all the details. Finding out how to make programming languages that did this took somewhere between four and ten years, and was a major boost to AI programming. The first major programming language to implement them was called *LISP*: it is still the most widely used AI language.

8.1 What is the goal?

PopBeast now has to start planning a sequence of actions to achieve this. The first step is to find out exactly what "open" means. This is done by looking up `opens` in a table which gives its meaning in terms of how it changes the world. In Eden, once you have opened a door, it disappears, i.e. the square in which it stood becomes empty. So the meaning can be expressed as "The effect of opening a door is that at the start of the activity there is a door in some square S, and at the end of the activity there isn't". In PopBeast's memory, this is stored as below:

```
verb( opens(D),
       [ square(D,S),door(D) ],
```

```
        [ clear(S) ]
     ).
```

Here, `verb` is a proposition which expresses information about other propositions. The first argument is the predicate which we used to represent "opens", the second is a statement about the start of the activity of opening, and the third is a statement about the end of the activity. `clear(S)` is how PopBeast represents the fact that square S is empty. If we can fill in the current values for `D` and `S`, we will have a description of our current goal — to convert a state in which square `S` contained a door to one in which it doesn't.

Note: as with world-models, not everyone believes that successful artificial intelligences will contain explicit representations of their goals. This view is descended from the idea that goals, beliefs and other "folk-psychology" constructs are a pre-scientific means we have developed for describing our own and other minds; they no more describe what really goes on than astrology describes how the cosmos operates. There are therefore no explicit representations of such things in human and animal minds: trying to find them is bad cognitive science. Furthermore, trying to build such representations into artificial minds is bad engineering. These views are put in *Taking Eliminative Materialism Seriously: A methodology for Autonomous Systems Research* by Tim Smithers, from *Towards a Practice of Autonomous Systems*, edited by Varela and Bourgine (MIT 1992: PSY KH:V 042; RSL M92.C00938); and in *An Emerging Paradigm in Robot Architecture* by Malcolm, Smithers and Hallam (AI box photocopy M149).

PopBeast *does* represent such entities — for teaching purposes, to illustrate classical AI. So going back to verbs, other ones could be expressed in the same way. The meaning of "go to object X" is stored as

```
verb( go(X),
        [ square(X,S)],
        [ square(X,S),square(me,S) ]
     ).
```

This expresses the fact that at the start of going, X is in a certain square called S, and at the end of going, PopBeast — `me` — is also in S.

8.2 Filling in referents

Before PopBeast can use the above information, it has to replace D and S below

```
[ square(D,S),door(D) ],
```

with a specific square and door. It does this by matching against the proposition

```
square(door(2), [8,4])
```

in its world model, giving S=`[8,4]` and D=`door(2)`. This process is called *finding the referent* of D and S. As I hinted in my discussion of "the", there are better ways to do this: PopBeast's method is a very poor approximation to what any decent language-understanding system should do, and to what we do. The problem of finding referents also comes up in interpreting pronouns, and was one of the objectives behind Shrdlu — consider a sentence like "Pick up the red block and put it on the large green box".

8.3 Planning

The `verb` table above has told PopBeast that opening something is equivalent to making it disappear from the square in which it was. With the referents of D and

S filled in, this implies that opening `door(2)` is equivalent to making it disappear from square `[8,4]`. How can PopBeast find a sequence of actions that will achieve this — a *plan*? This is called *planning*, and the program that does it is called a *planner*.

Remember that in each turn, PopBeast can perform one of eight primitive actions: left, right, forward, back, drop, grab, use, and wait. To reason about what sequence of actions will achieve a goal such as `clear([8,4])`, PopBeast needs to represent the effect of each of these actions. There are various ways to do this; the one I have chosen is based on that used in an early planner called *STRIPS*.

The idea is that we represent the state of the world at any time by a set of propositions, as PopBeast does in its propositional world-model. We then represent each action by a table showing how, if it were carried out, it would change the state. In particular: what does it make false that used to be true; and what does it make true that used to be false?

Let's look at "grab" as an example. Before PopBeast grabs an object, its hand must be empty, and the object must be in the same square as PopBeast. After grabbing, the hand will contain the object, and the square will be empty. We can represent this by saying that when grabbing X in square S, PopBeast makes this fact true:

```
have(X)
```

and makes these facts false:

```
square(X,S), empty
```

Here, `have(X)` means "I am holding X", and `empty` means "I am holding nothing".

Other actions can be specified in the same fashion. When using a key K on a door D in square S, the facts

```
empty, clear(S)
```

become true, and the facts

```
have(K), square(D,S)
```

become false. (Remember that after using a key on a door, the door disappears from its square, and the key from PopBeast's hand.) This way of describing actions is called the *add-list/delete-list representation*.

We need one further piece of information for each action: under what conditions it can be performed. You can only open a door if you are in the same square as it and you have a key; you can only grab an object if your hand is empty and you are in the same square as it. These necessary conditions are called the action's *preconditions*. We represent these in the same way as the add- and delete-lists: as sequences of propositions. Thus the precondition for grabbing X in square S is

```
square(X,S), empty, square(me,S)
```

which literally translated from the logic, means "X is in S, and PopBeast is in S, and PopBeast's hand is empty".

Similarly, the precondition for opening door D with key K in square S is that

```
have(K), square(D,S), square(me,S)
```

which means "I am holding K, D is in square S, and I am also in square S".

The add-lists, delete-lists and preconditions are collected together for each primitive action, in a table rather like the `verb` table. As an example, here are the descriptions of the primitive actions `grab` and `use`:

```
action( grab(X,S),
        [ square(X,S), empty, square(me,S) ],
        [ square(X,S), empty ],
        [ have(X) ]
      ).

action( use(K),
        [ have(K), square(D,S), square(me,S) ],
        [ have(K), square(D,S) ],
        [ empty, clear(S) ]
      ).
```

8.4 Building plans from actions

Having decided to represent actions as above, how do we chain them together so as to achieve a specified goal? The simplest method is *blind search*; this is like playing a game of chess by considering all possible moves from your current board state, then all possible moves from each of the resulting states, then all possible moves from *those* resulting states, and so on. Essentially, we draw a tree of possibilities, form all the branches, and pick the one which leads to the desired goal. I won't draw such a tree here: see page 138 of *Cognitive Science projects in Prolog* by Scott and Nicolson (LEA 1991: PSY KH:S 528; RSL M92.C00948) which shows such a tree for the blocks world.

This approach rapidly becomes unfeasible. The number of possibilities increases so fast with level in the tree, that we run out of processor time, no matter how fast our computer. This explosion of possibilities is called the *combinatorial explosion*. In the Foreword to *Computers and Thought*, Sloman calls AI the "science of productive laziness" — can we be lazy and avoid an exhaustive search?

8.5 Means-end planning and goal regression

An idea introduced by Newell, Simon and Shaw in their *General Problem Solver* (GPS) program was that instead of choosing actions blindly, as above, the planner should concentrate on those actions which are most likely to lead it towards the final goal. This is called *means-end* planning: choosing a means which is likely to achieve the desired end.

My planner uses a method called *goal regression*, and introduced by STRIPS. The idea is that we look at the add-lists of all the actions. Some actions will have add-lists which add propositions we want in our goal. We imagine that we'll apply one of these actions, and ask what new goals need to be satisfied in order for us to do so. We then treat these in the same way and pick an action that will satisfy them, and so on.

An example will help. As I showed in section 8.1, our goal state is

```
clear( [8,4] )
```

We can select actions that might work towards this goal by looking at the add-lists. In essence, we want an action whose add-list contains a `clear`: that is, one which makes a square clear. One such action is to use a key. Its add- and delete-lists show that we would get the state

```
clear( [8,4] )
```

if we applied it to the following state:

```
have( key(1) ), square( door(2), [8,4] ).
```

In other words, we can make [8,4] clear if we have key 1, and we use it on a door in [8,4]. But this requires us to have a key, so we set that as a new goal. We now need to find an action which gets a key into our hand: namely grab. But that requires us to be in the same square as a key, so we must set the goal of getting to that square. And so on. What we are doing is simplifying the problem by breaking the original goal into a series of *subgoals*; in this case, using goal-regression to do so.

8.6 Planning at a level of abstraction

When my planner had finished, it produced the following plan:

```
[ go([4, 4], [5, 6]),
  grab(key(1), [5, 6]),
  go([5, 6], [8, 4]),
  use(key(1))
]
```

meaning "go from [4,4] to [5,6]; grab key 1; go from [5,6] to [8,4]; use key 1". You will see that this plan is incomplete; it doesn't say anything about *how* to go between two points. The reason for this is that my planner is not very efficient (though better than blind search), and it runs slowly if it has to consider too many details. So I have treated "go" as a primitive action: as far as the planner is concerned, one can always go between any two points. This is called *planning at a level of abstraction*: making a plan whilst ignoring some of the details.

Before PopBeast can obey this plan, it must be fleshed out with the actual moves. To do this, PopBeast runs through, looking at all the gos. For each one, it plots a short path between the two points, and then converts that into a sequence of lefts, rights, forwards, and backs. It could be that no such path exists, if (say) there is a wall between the points. In that case, the path-plotter would have to tell the planner to make a new plan. In my example, paths do exist, and I'll ignore situations where they don't.

8.7 Fleshing out plans

We find paths using the geometric world model. Consider the problem of going from [4,4] to [5,6]. We start at [4,4] and enumerate all the clear neighbours. We then calculate the distance from each of them to [5,6], using the standard formula. We pick the neighbour for which this distance is least, and then repeat, finding its neighbours, calculating distance, and so on. This is called *guided search*. Instead of picking a neighbour blindly, we pick one which seems most likely to lead to our goal. Our measure of success — our *evaluation function* — is distance.

In this particular problem, this gives us a nice short path. However, it wouldn't do so if there had been an obstacle a bit later on: our use of distance as a guide would not tell us about this. Picking the neighbour with the least distance from the goal is a good rule-of-thumb — a good *heuristic* — but one that sometimes fails.

After the search had finished, it generated this path:

```
[ [4, 4], [4, 5], [5, 5], [5, 6] ]
```

The final stage was to convert it into moves. This needed to know PopBeast's initial heading, and to keep track of how it changed. The result was this set of moves:

```
[left, forward, right, forward, left, forward]
```

The other path was generated in the same way, giving a complete plan of

```
[left, forward, right, forward, left, forward,
 grab,
 right, forward, forward, right, forward, forward, left, forward,
 use
]
```

8.8 Planning and execution

Having made a plan, PopBeast follows it an action at a time. There is nothing in the current PopBeast that allows it to adjust if an action fails, or if the world changes in the meantime. In general, the early planners assumed that these problems didn't arise. More recent work, especially in robotics, has had to face them.

9 The symbol-grounding problem; and a warning

Most of the symbols in PopBeast have names which look like English words: `opens`, `assertion`, `in`. When looking at programs that use such symbols, it's very easy to fall into the trap of thinking that they mean a lot more to the program than they do. When *we* see the symbol `opens`, it induces a rich network of concepts and associations. None of these, however, are available to PopBeast: to it, `opens` only gains meaning by virtue of the way it is manipulated by various parts of the program.

Another way to make this point is that PopBeast would behave exactly the same if I were to systematically replace every name by its equivalent in some other language. I could, for example, replace `square` by `vierkant`, `key` by `sleutel`, `me` by `mij`, `door` by `deur` …As long as I have done this so that names are different in this new language wherever they were different in the original, the program won't suffer. I could of course also use made-up names: `square` by `z`, `key` by `zz`, `me` by `zzz`, `door` by `zzzz` …

This is an important point, well worth bearing in mind when reading AI programs and books. Most of the examples of symbolic representations you will see use English names, for the simple reason that authors and programmers find it easier to work with names with familiar connotations. These names mean so much to us that we can easily forget they have no such meaning to their program. Beware of this. I recommend reading *Artificial intelligence and natural stupidity* by Drew McDermott from *Mind Design* by Haugeland (MIT 1981, PSY KH:H029) as a preventative. It assumes some experience with semantic nets, so you may want to come back to it when you've read about those.

If the internal symbols don't gain meaning from their names, how do they gain it? PopBeast's brain does not operate in isolation, but perceives and acts in what, to it, is an external reality. How does the internal symbol `door` come to be connected with what PopBeast perceives when it sees a # in Eden? This is the *symbol-grounding* problem: how, in an artificial or natural symbol-manipulating system, the internal symbols come to be connected to external referents and actions. In PopBeast, the answer is that although most of the symbols' names are completely arbitrary, this is not true of the images on PopBeast's retina, nor of the motor actions that it obeys. These are determined by Eden's "laws of physics", and somewhere inside PopBeast there is a consistent mapping between them and symbols like `door` — a mapping set up by me. The symbol-grounding problem becomes more acute in learning systems which must identify for themselves novel features and properties for which their programmer has not provided pre-defined symbols.

10 The problems of scale

To close this handout, I should mention that it is now evident that very few, if any, of the subprograms (such as the planner) built into PopBeast will scale up to larger problems.

To digress for a bit, it's interesting to note that there are precedents for such scaling problems in nature: the worst of Hollywood aside, a man-sized flea would not be able to jump over the Empire State Building. Insect respiration relies on oxygen being able to diffuse from pores in the exoskeleton to every part of the body, and this is only possible for small insects. Above a certain size, a complete redesign — lungs, with their fractally enormous surface area — was needed. [1]

Perhaps it's not therefore surprising that AI methods which work on a very small scale should fail at a larger one. In fact, there is now an entire branch of computer science devoted to the way in which the time and space requirements of algorithms scale with the size of a problem.

And it's important not to underestimate the difference in scales between microworlds and the real world. A typical AI microworld, Eden included, contains perhaps ten to fifty "objects", where all objects of the same type are identical. Generally, they have no internal structure, no parts, and at most ten different attributes. In addition, the AI inhabiting such a microworld typically receives complete perceptual information about its environment. Finally, there is usually nothing corresponding to friction, weather, predators, or anything else which can cause the environment to behave unpredictably.

11 So what has classical AI achieved?

Firstly, note that I chose the subprograms I have described because they were free and could be used without modification. This generally means they were also old, and written as a demonstration of some concept rather than as a robust and reliable system for heavy-duty use. In some cases, there are relatives of these subprograms that do a lot better. For example, one can obtain heavy-duty "industrial strength" planners that have proved themselves indispensable for planning industrial assemblies. If one only wants to solve single tasks, the let's-study-a-single-task approach of classical AI can be rather useful!

Some of these methods — most of the ones used to implement expert systems, for example — only work within extremely restricted domains, but are useful as long as we recognise their limitations.

Note also that recent research is producing new methods which may succeed where the old ones failed. One extremely useful application is "inductive machine learning", which involves constructing general logical rules from particular examples. This is very promising for drug design, amongst other areas.

Secondly, as far as classical AI's contribution to psychology goes, it has made the subject more rigorous, by demonstrating how much is left unspecified in a theory which is not precise enough to be runnable on a computer.

It has helped make the idea of mental states respectable, shown that we should ask in detail *how* the mind does things as well as *what* it does, and demonstrated the kinds of language in which we can talk about mental processes. Some of these may need to be modified, but in their newer incarnations in the various forms of non-classical AI, they are perhaps not as easy to pick up as in their older, possibly wrong, but more precisely specified forms.

[1] A nice essay on the problems of scale, including an application to the design of churches, is given in the chapter *Size and Shape* from *Ever Since Darwin* by Stephen J Gould.

It has given us a number of general ideas about how computational systems can be organised, and terminology for describing their properties. These include the ideas of representation, control, algorithmic complexity, and levels of description, as well as notions about how to connect small computational systems to build large ones.

(I would like to trace the history of computer vision here, since Marr's work on early vision is one of the most well-known contributions of AI to psychology. I probably wouldn't call it classical AI, but it certainly used the notion of representation, for example. It would be interesting to see how it arose, if at all, from some of the wrong turnings taken by earlier researchers, right back from the alleged task Minsky set a graduate student in 1960: "go and solve computer vision as a summer project". Of course, Marr is by no means the last word in vision.)

There are other views of AI's contributions. Sometime during the course, you should look at *The Foundations of AI*, edited by Partridge and Wilks (CUP 1990; PSY KH:P 025). It contains a few very technical papers on subjects like theorem-proving, but also a number of general ones by Marr, Boden and others, giving their views of AI.