Contents

1 Production systems and working memory **2**
1.1 Holidays 2
1.2 Another example 4

2 Production systems, predicates, and logic **4**
2.1 Predicates and arguments 6
2.2 What do they mean? 6
2.3 Skiing again 7
2.4 Rules, inference, implication, and logic programming 7

Production systems (2)

Jocelyn Paine

June 17, 1997

1 Production systems and working memory

In the previous handout on *Production systems*, I introduced the general idea of a production system, and then demonstrated one particular type of production system — one functional architecture of bug cognition — applied to two agents. In some ways, the demonstration was a bit of a cheat. I introduced production systems as working mainly on symbols in a short-term memory, but the two agents didn't use STM at all. They worked entirely on perceptual predicates and actions, and their STMs remained empty throughout. In this handout, I'll remedy that, with agents that think a bit before reacting.

1.1 Holidays

In order to think, you need something to think about. In the context of Eden, this means that you must be able to gather data before being able to use it. This introduces issues of learning, navigation, and so on, which I don't want to complicate matters with. I'm therefore going to start with an example that has very little to do with Eden.

Please look at the rules below, in figure 1.

```
resolve [r].


stm [ mp,
      hot
    ].


stm_predicates all.


computer_hacker => rich.
management_consultant => rich.
mp => rich.

hairdresser => poor.
teacher => poor.
toilet_cleaner => poor.
```

```
poor => near.
rich => far.

near, hot => costa_del_sol, done.
near, cold => aberdeen, done.
far, hot => egypt, done.
far, cold => switzerland, done.
```

Figure 1 Where to go for holiday.

In this program, we have the statement `stm_predicates all`. Like the `resolve` statement, this is something that programmers call a *declaration* — a statement that isn't itself obeyed as the program runs, but that alters the way other symbols are interpreted or other statements are executed. The declaration `stm_predicates all` tells Eden that all the names in rules refer to symbols in STM and not to perceptual predicates or motor commands. Actually, there is one exception to this: this is `exec` which is always taken as obeying an action.

The initial contents of STM is given by the symbols after `stm`: that is, `mp` and `hot`.

What do these symbols mean? The production system does not care, and would work equally well if they were all meaningless names. However, you can see that they have something to do with choosing holidays. The idea is that you start the STM off by telling it what your job is, and whether you like hot or cold weather. The job must be one of `computer_hacker`, `hairdresser`, `management_consultant`, `mp`, `teacher`, or `toilet_cleaner`. The weather must be either `hot` or `cold`. The production system then runs a few rules, and eventually comes up with the name of a country or resort, which it puts into STM.

You can test the production system for yourself by selecting `holiday.rules` from the `Compile` dialogue and then running it.

How do the rules work? I hope this is clear from my original discussion of production systems. If not, you can follow it by looking at the trace areas. I've printed below a record of how the STM changes.

```
hot (0.2)
mp (0.1)

rich (1.1)
hot (0.2)
mp (0.1)

far (2.1)
rich (1.1)
hot (0.2)
mp (0.1)

done (3.2)
egypt (3.1)
far (2.1)
```

```
rich (1.1)
hot (0.2)
mp (0.1)
```

Figure 2 The holiday PS's STM.

If you want to experiment with changing these rules, or the initial contents of STM, you may. Call up the text editor, Open the appropriate file (its name must end in `.rules`), and edit and save it. Then do Compile.

1.2 Another example

Have a look at the rules in `holiday2.rules`, and try running that.

2 Production systems, predicates, and logic

In thes holiday examples above, all the rules' primitive conditions were simple "words" like `hot` and `mp`. It is possible to have more complex conditions, which is where a connection with logic comes in. For an example, try `skier.rules`:

```
stm [purpose(fun),
     pressups(50),
     lessons(173)
    ].

resolve [r].

pressups(N),N>25 => fitness(good).
pressups(N),N=<25 => fiitness(poor).

fitness(poor), lessons(L),L<10 => rating(beginner).
lessons(L), L>=10 => rating(advanced).
fitness(good) => rating(advanced).

purpose(fun), rating(beginner) => resort(st_sartre),done.
purpose(serious), rating(beginner) => resort(schloss_heidegger),done.
purpose(fun), rating(advanced) => resort(wittgenstein_gladbach),done.
purpose(serious), rating(advanced) => resort(chateau_derrida),done.
```

Then try `sweets.rules`:

```
resolve [r,s].

stm [
   considering(jim),

   bank_balance(jim,20.00),
   bank_balance(john,20.00),

   daddy(jim,stockbroker),
   daddy(john,toilet_cleaner),
```

```
   likes(jim,lumpfish_caviar),
   likes(john,mars_bar),

   in(shop,mars_bar),
   in(shop,lumpfish_caviar),

   costs(mars_bar,0.35),
   costs(lumpfish_caviar,50.00)
    ].


considering(P),
in(shop,Item),
likes(P,Item)
=>
will_buy(P,Item).

considering(P),
will_buy(P,Item),
costs(Item,Price),
bank_balance(P,Owns),
Owns > Price
=>
will_be(P,in_credit).

considering(P),
will_buy(P,Item),
costs(Item,Price),
bank_balance(P,Owns),
Owns =< Price
=>
will_be(P,overdrawn).

considering(P),
will_be(P,overdrawn),
daddy(P,stockbroker)
=>
will_be(P,ok),done.

considering(P),
will_be(P,overdrawn)
=>
will_be(P,not_ok),done.

considering(P),
will_be(P,in_credit)
=>
will_be(P,ok),done.
```

Can you see how the `considering(jim)` in STM focusses the system's attention on what happens to Jim, rather than John?

2.1 Predicates and arguments

In fact, the structure of the rules will be familiar if you have studied logic. In most places now, we have groups of symbols between commas, rather than just single symbols. A group starts with a name followed by an open bracket. This is called the *predicate*. After the bracket come a sequence of names or numbers separated by commas. These are called the *arguments* of the predicate. Finally, there comes a closing bracket. Note that the word "argument" is used in a technical sense peculiar to mathematics; this sense has nothing to do with its everyday meaning.

Predicates can have any number of arguments, so groups like these are all valid:

```
job(mp)
loves(john,mary)
pub(prince_of_wales,walton_street,oxford,morrells)
```

If a predicate has no arguments, then you just write its name:

```
mp
hot
```

Prolog will not allow you to have the name followed by an empty pair of brackets, though some other languages do.

2.2 What do they mean?

I have repeatedly emphasised that, as far as the production system is concerned, the groups of symbols are completely arbitrary. Only their form matters to the program that runs the matching, resolution, and firing processes, not their content. However, when we write programs, we usually do intend each name to have some content — to refer to something. Here are some examples of two-argument predicates, where the meaning should be clear.

```
event( 1756, mozarts_birth )
loves( joe, mary )
teaches_at( joseph, st_annes )
colour( grass, green )
legs( sheep, 4 )
costs( mars_bar, 0.35 ),
is_a( sheep, mammal )
```

For such predicates, the predicate usually names a relation that holds between its arguments. This is easiest to see with spatial predicates, such as

```
on( block_a, block_b )
```

You can also use predicates to say that an object has a certain property, to say that an object belongs to a certain class, or to give the result of operating on the object in a certain way. The last use is like that of a function in maths, where one argument is the input and the other is the result.

Predicates of fewer or more than two arguments can also be interpreted as relations, though this is less obvious to non-logicians.

In most programs, the names appearing as arguments are used in one of three ways:

To name individuals	`john`	`egypt`
To name properties	`rich`	`hot`
To name classes	`teacher`	`wine`

A predicate and its arguments form a *proposition*: a logical statement. This may be either true or false, depending on the meaning of all the names and the state of the real world. Here are some propositions and possible meanings:

`mp`	I am an MP.
`egypt`	I should go to Egypt on holiday.
`rich`	I am rich.
`loves( john, mary )`	John loves Mary.

When writing a program, you should always keep the intended meanings of your predicates and arguments in mind. It is not always clear from the names alone. For example, does `gold` refer to gold as a chemical entity in general, or to gold in finance at a particular time, or to a particular chunk of gold? Compare its meaning in the sentences "Gold does not dissolve in water", "Gold rose by £2.45 today", and "The gold in this ring was mined in 1847".

As another example, of a different kind, does `on(block_a,block_b)` mean that A is standing on B, or that B is standing on A? Be careful about the order of your arguments.

2.3 Skiing again

Having read the above, please write down what you think the predicates in `skier.rules` and in `sweets.rules` mean. Do so in a little table, like this:

`fitness(PG)`	PG: a rating	The skier's fitness is PG.
`costs(X,Y)`	X:a sweet, Y:a price in pence	One X costs Y.

In the first column, put the name of the predicate and a variable for each argument. In the second, put what kind of thing the arguments are (men, prices, sweets etc). In the third, put an English sentence describing the predicate's meaning.

2.4 Rules, inference, implication, and logic programming

When we think of groups of symbols as propositions, we can see that a production rule is a device which, given the fact that its left-hand propositions are true, makes explicit the fact that the right-hand propositions are true. We can therefore view production rules as *rules of logical inference*: rules which, given one set of true propositions, *infer* another set of true propositions. One of the rules `skier.rules` infers the proposition `rating(advanced)` from the proposition `fitness(good)`. That is, if you tell the rule that `fitness(good)` it will infer that `rating(advanced)`.

This is what *logic programming* is all about. In languages such as Pascal and Basic, you program by telling the computer how to accomplish some task. Your program consists of a sequence of little instructions, which if obeyed in the right order will (hopefully) do the right thing.

But in logic programming, you describe your program in logic. Your program consists of an initial set of propositions, plus a set of rules which infer new propositions from old. In our production system, the initial set of propositions is the initial STM, and the rules are the production rules.

Logic programming became famous at the start of the 80's, with the growth of interest in expert systems, Prolog, and the Japanese Fifth Generation Project. For background reading on this, see *The Guide to Expert Systems*, by Alex Goodall (Learned Information, 1985; somewhere in PSY). Look at the chapters on logic programming (number 6, I think), and on the history of AI. The most famous logic programming language is Prolog, and in fact the Eden production system is built on Prolog. They therefore share a number of similarities, particularly in the syntax of predicates and in the way conditions are handled.

One of the important things about logic programming is that you can view the rules, as well as the propositions, as logical statements. The rule

```
fitness(good) => rating(advanced).
```

can be read as an implication: "If my fitness is good, then my rating is advanced" or ""My fitness is good" implies "my rating is advanced"". Similarly, the rule

```
far, hot => egypt, done.
```

can be read as the implication "If you can go somewhere far, and you like it hot, then Egypt would suit you".

You should note that not all production systems can be used for logic programming. The Eden one can because it has facilities for grouping symbols into predicates and arguments, and because the matching and rule-firing were designed (with help from Prolog) to model logical inference. However, matching and rule-firing don't have to work this way.