Appendix A – An example of constructing a model for the simple game of Jugs.

The game of jugs is formulated as follows: There are two jugs of known quantities that have no intermediate markings on them. The aim of the game is to achieve a specific quantity of water in one of the jugs. The only permissible operations are to empty or fill a jug or pour water from one jug to the other such that the destination is full or the source is empty.

To aid in the elaboration of the model construction each of the lines of the model have been numbered, although these numbers are not part of the script. To begin we can define the capacities of the two jugs, which we will denote A and B respectively.

```
1. %eden
/* set some capacities of the jugs */

2. capA = 5;
3. capB = 7;
```

We use the line drawing notation to specify the display of the jugs. The size of the jugs has a fixed width in this model and a height that is dependent on the capacities of the jug.

```
/* develop an interface for the jugs */
4. %donald

5. int scalefactor
6. scalefactor = 100

7. point Abotleft, Abotright
8. line Abase, Aleft, Aright
```

```
Abotleft = {100,100}
Abotright = Abotleft + {300,0}
Abase = [Abotleft,Abotright]
Aleft = [Abotleft, Abotleft+{0,capA!*scalefactor}]
Aright = [Abotright, Abotright+{0,capA!*scalefactor}]

point Bbotleft, Bbotright
line Bbase, Bleft, Bright

Bbotleft = Abotleft + {400,0}
Bbotright = Bbotleft + {300,0}
Bbase = [Bbotleft,Bbotright]
Bleft = [Bbotleft, Bbotleft+{0,capB!*scalefactor}]
Bright = [Bbotright, Bbotright+{0,capB!*scalefactor}]
```

In the DoNaLD notation, variables have to be declared and are strongly typed unlike in EDEN. The ‘=’ sign in DoNALD has the same functional equivalence as the `is` operator in EDEN in maintaining dependencies. The reference points of the jug display are dependent on the base point of the jug (`Abotleft`). The position of jug B is dependent on jug A. The variable `scalefactor` is used to scale the display of the jug heights to be visible. The ‘!’ operator is used to reference EDEN variables in DoNaLD. Statements are terminated by line breaks unlike EDEN which uses the ‘;’ symbol

```
%eden

/* initial contents of the jugs */

contentA = 0;
contentB = 0;
```

We can now define the amount of water in the jugs. Note that comments in the EDEN notation are between the /* */ symbols.

```
/* some information about the jugs */

24. Afull is (capA==contentA);
25. Bfull is (capB==contentB);
26. Aempty is (contentA==0);
27. Bempty is (contentB==0);
```

These definitions are commonsense observations of the jug situation. A jug is only full if its content is equal to its capacity and is empty if it has no water. These conceptual observables define information we can ascertain by looking at a jug.

```
/* now we can define the surface of the liquid */

28. %donald

29. line Asurface
30. Asurface = [Abotleft+{0,contentA!*scalefactor},
Abotright+{0,contentA!*scalefactor}]
31. line Bsurface
32. Bsurface = [Bbotleft+{0,contentB!*scalefactor},
Bbotright+{0,contentB!*scalefactor}]
```

We have now added a water surface to the jugs model in the line drawing. The surface is represented as a line from the left side of the jug to the right side of the jug. Its height is dependent on the content of the jug.

```
%eden

/* now we define some jug operations */

proc fillingA {
  if (!Afull) {
      contentA++;
      eager();
      fillingA();
  }
}

proc fillingB {
  if (!Bfull) {
      contentB++;
      eager();
      fillingB();
  }
}

proc emptyingA {
  if (!Aempty) {
      contentA--;
      eager();
      emptyingA();
  }
}

proc emptyingB {
  if (!Bempty) {
      contentB--;
      eager();
      emptyingB();
  }
}

proc pouringAtoB {
  if ((!Bfull)&&(!Aempty)) {
      contentA--;
      contentB++;
      eager();
      pouringAtoB();
  }
}

proc pouringBtoA {
  if ((!Afull)&&(!Bempty)) {
      contentB--;
      contentA++;
      eager();
      pouringBtoA();
  }
}
```

These six actions are the important state transitions. They conform to the basic operations in the jugs game, namely emptying, filling and pouring. They can be tested by running them in the input window with commands of the form ‘`emptyA();`’. Each procedure runs as a stepwise operation since we cannot (in practice) determine the current content of a jug by inspection if it is partially full. Each step consists of checking whether an enabling condition is satisfied, making a stepwise change to the contents, updating the display and then continuing by recursively calling the same action.

```
/*now develop a scout interface for the main operations*/

%scout

window fillA = {
  type: TEXT
  string: "Fill A"
  frame: ([{10, 10}, {80, 30}])
  border : 1
  sensitive: ON
};

window fillB = {
  type: TEXT
  string: "Fill B"
  frame: ([{90, 10},{160, 30}])
  border : 1
  sensitive: ON
};

window emptyA = {
  type: TEXT
  string: "Empty A"
  frame: ([{170, 10}, {240, 30}])
 border : 1
 sensitive: ON
};

window emptyB = {
  type: TEXT
  string: "Empty B"
  frame: ([{250, 10}, {320, 30}])
104.   border : 1
105.   sensitive: ON
106. };

107. window pourAB = {
108.   type: TEXT
109.   string: "A -> B"
110.   frame: ([{330, 10}, {400, 30}])
111.   border : 1
112.   sensitive: ON
113. };

114. window pourBA = {
115.   type: TEXT
116.   string: "B -> A"
117.   frame: ([{410, 10}, {480, 30}])
118.   border : 1
119.   sensitive: ON
120. };
```

The lines 78-120 define interface windows to activate the emptying, filling and pouring operations. The user does not then need to know the procedure names but can perform operations using a graphical user interface. The Scout notation is used to define the windows. Each window is sensitive to mouse button events (sensitive: ON). When the mouse is clicked in a window a definition for that mouse event is created. This can be used to trigger an action based on that variable's new value.

```
121. display operations = <fillA/ fillB/ emptyA/ emptyB/
pourAB/ pourBA>;

122. screen = operations;
```

Lines 121 and 122 group the set of windows into a display and then set the screen to display those windows on the interface, as shown in Figure A.1.

```
123. %eden

/* now we can define some routines to be triggered on the
mouse click on a window */

124. proc filljugA : fillA_mouse_1 {
125.      if (fillA_mouse_1[2]==5) { /*on button
release*/
126.           fillingA();
127.      }
128. }

129. proc filljugB : fillB_mouse_1 {
130.      if (fillB_mouse_1[2]==5) { /*on button
release*/
131.           fillingB();
132.      }
133. }

134. proc emptyjugA : emptyA_mouse_1 {
135.      if (emptyA_mouse_1[2]==5) { /*on button
release*/
136.           emptyingA();
137.      }
138. }

139. proc emptyingjugB : emptyB_mouse_1 {
140.      if (emptyB_mouse_1[2]==5) { /*on button
release*/
141.            emptyingB();
142.        }
143. }

144. proc pourAB : pourAB_mouse_1 {
145.        if (pourAB_mouse_1[2]==5) { /*on button
release*/
146.            pouringAtoB();
147.        }
148. }

149. proc pourBA : pourBA_mouse_1 {
150.        if (pourBA_mouse_1[2]==5) { /*on button
release*/
151.            pouringBtoA();
152.        }
153. }
```

The lines 124-153 define actions to be taken on mouse button presses in the windows. Definitions of the form ‘window’_mouse_1 are created each time a mouse event is received in a sensitive window. It is a list specifying the button that has been pressed or released and the window coordinates at which it has been pressed. The second element of the list has a value of 5 when the button is released and this condition causes the action to be executed. We have now constructed a working model that allows us to play the game of jugs.

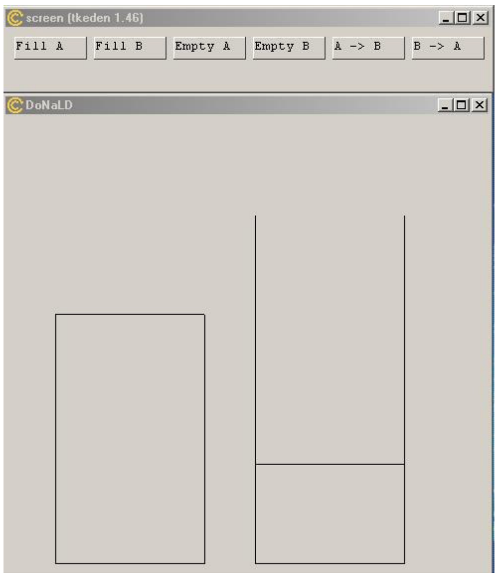


Figure A.1 – The simple jugs model

Appendix B – An example of building a parser

This appendix is taken from Antony Harfield's final year project on the agent-oriented parser [Har03]. It includes information about the agent-oriented parser that is essential for understanding the discussion in section 5.4.1.

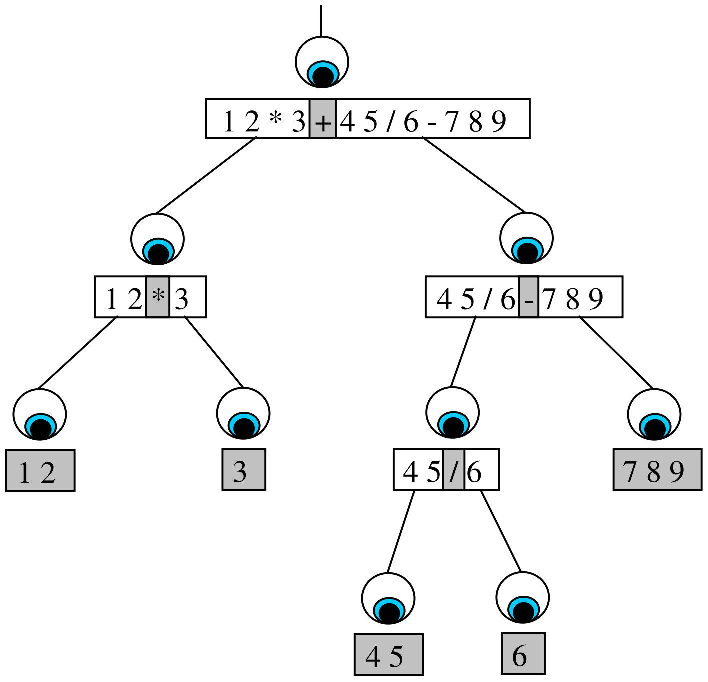
Getting Started

Parser concepts

In order to learn to use the Agent-oriented Parser in Eden we must put aside our prior knowledge of parsing. Conventional parsers read each input character, one at a time, and not until the entire string is read can any meaning be derived. Instead, it is useful to think about how we, as humans, read languages (natural or otherwise). When we read sentences, we do not remember each character, or even each word, but our brains register the important words. From this we are able to derive meaning.

An agent is an independent entity capable of acting on and interacting with an environment. In the Agent-oriented Parser, the agents have a set of rules (a grammar) with which to parse any input. Each agent will take an input string and a rule, with which it will determine whether the rule can be applied. If the rule is applied, then more agents could be generated to work on substrings. The agent will fail if it cannot apply the rule.

A rule specifies a string that must be observed in the input string. For example, an agent might have the input '1+2' and the most important string it must find could be '+'. When the agent observes a match, it will pass the remaining substrings (if any) to new agents. These are called child agents.



The diagram shows how agents could parse a string. In this example the input is an arithmetic expression and the rules are such that the expression is parsed using standard operator precedence.

Writing notations

A notation is defined as a set of rules, which specify the behaviour of our agents. A rule is simply an Eden list. The basic template for a rule:

```
myrule = [ operation, pattern, [ rule, ... ], [ "fail", rule ]
      ];
```

An `operation` is a string containing the name of the operation an agent should perform. These will be explained in detail later. A `pattern` defines what string the agent is trying to find (or observe) in the input. A `rule` is a string containing the name of a rule (a variable name of an Eden list).

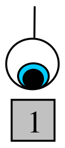
The first item in the list is always an operation. The second item is always the string to be matched. The third item is optional, it is a list of rules to apply to the resulting substrings – the number of substrings depends on the operation to be performed. This is the minimum that a rule can contain. It is likely that most rule definitions will have a fail clause, such that if the agent fails then it will try another rule. The fail clause is a two item list, the first item is always the string "fail" and the second item is a string containing the name of a rule.

Developing a calculator notation

A simple calculator is able to accept digits and arithmetic operators. It will calculate the result of the input expression. Calculators also allow nested expressions using brackets.

Start with a simple model

Recall that our Agent-oriented Parser uses an agent to observe a string in the input. The simplest agents therefore match the input entirely. On a calculator, the user can input just a digit, and the result will be that same digit.



The first operation we will be introduced to is "literal". This operation attempts to match the entire input string (see diagram right). It does not create any child agents.
Lets define a rule that matches the digit ‘1’:

```
number1 = [ "literal", "1" ];
```

To invoke the parser, you must supply an input string and a starting rule. The only input our rule should match is ‘1’, so to test this run the parser:

```
dfparse("1", "number1", []);
```

It should accept. The first parameter is the input and the second is the starting rule. Try other inputs to get the parser to reject.
Our language is not very powerful, being only able to accept the digit ‘1’. Now we will introduce the fail clause. We can get our parser to try another rule should the operation fail:

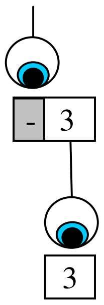
```
number1 = [ "literal", "1", [ "fail", "number2" ] ];

number2 = [ "literal", "2" ];
```

The language will now accept either of the digits ‘1’ or ‘2’. We could extend this method to accept all the digits.

Iteratively developing the model

Now that our calculator can parse digits, we will introduce operators to the input. This is an important concept in Empirical Modelling, the ability to build up a model in an iterative fashion.

Our current model allows us to parse positive numbers, so an obvious extension is to allow negative numbers too. We can define an integer as a number with an optional minus symbol (-) in front of it.



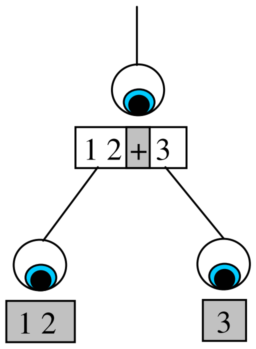
The next agent operation we shall introduce is "prefix". This operation matches a string at the beginning of the input. If a match is made then a child agent is created with its input as the unmatched part of the agents input (see diagram right). This operation can be used to detect a minus symbol at the beginning of our number:

```
term = [ "prefix", "-", "number1", [ "fail", "number1" ] ];
```

Notice that we use the same rule for the child (third item) and the fail clause. The parser will try to match a minus sign at the start of the input, if it matches then it will match the remainder of the input as a number, else it will match the entire input as a number.

A similar operation is "suffix" which does exactly the same as "prefix", but at the end of a string. An example of using suffix is to remove the semi-colon from the end of a string (i.e. to parse a statement in C/Java/Eden)

The parser should now accept any positive and negative numbers. We shall now look at how we can parse arithmetic operations (e.g. +, -, *, /). As always we will begin with something simple and build the model up. We will first try to parse expressions containing only additions of terms, where our terms are the integers we learnt to parse above.



This is where the most powerful agent operation comes in. The "pivot" operation searches the input (left to right) for a specified string. If it finds a match, then it creates two child agents, one for the left substring and one for the right substring (as shown in the diagram on the right). Our parser would pivot on the addition sign:

```
expr = [ "pivot", "+", "expr", "expr", [ "fail", "term" ] ];
```

An "expr" agent is looking for an addition sign, and if it finds one it creates two children that also search for expressions. If the agent finds no addition sign on the input, then the fail clause specifies that the input must be a term.

In order for our parser to recognise expression containing other operators it is necessary for us to have a rule for each string to be matched:

```
expr = [ "pivot", "+", "expr", "expr", [ "fail", "expr2" ] ];

expr2 = [ "pivot", "-", "expr", "expr", [ "fail", "expr3" ] ];

expr3 = [ "pivot", "*", "expr", "expr", [ "fail", "expr4" ] ];

expr4 = [ "pivot", "/", "expr", "expr", [ "fail", "term" ] ];
```

Notice that we search for operators in their reverse precedence order. This can be explained by taking an example, say the input is ‘1+2*3’. First we would pivot on the addition sign giving us two substrings ‘1’ and ‘2*3’. We have broken the calculation down into two sub-calculations which we will add together later. The deepest level will get calculated first, which in this example is ‘2*3’. Therefore we are observing the rules of precedence correctly.

Regular expressions

Using just the pivot and literal operations gives you all the power you need to develop languages. However, the rules would be quite cumbersome without regular expressions. The Agent-oriented Parser has 3 other operations that deal with basic regular expressions.

The first is "read_all", which is a regular expression version of the "literal" operation. This will attempt to match every character in the input string with a set of characters specified in the rule. For example, the following will match any number using a single rule (compare with our inefficient earlier method):

```
number = [ "read_all", [["0","9"]] ];
```

The second item in the list is a list of tuples. The tuples define the range of characters included in the set to be matched (inclusive). Note that the "read_all" operation accepts the empty string.

The other two regular expression operations are "read_prefix" and "read_suffix", which operate in much the same way but you also specify the number of characters to attempt to match. For example, the following will match one letter of the alphabet at the beginning of the input string:

```
letter = [ "read_prefix", [[["a","z"],["A","Z"]],1], "nextrule"
      ];
```

Perl-style regular expressions

The above regular expressions lack power, for example, you cannot specify rules for real numbers or identifiers. Our definition of a "number" will accept the empty string – not desirable in most cases!

Three more regular expression operations exist in the latest version of the Agent-oriented Parser. These are "literal_re", "prefix_re" and "suffix_re". The first matches the whole input, where as the other 2 match the beginning and the end of the input respectively. The pattern to be matched is a perl-style regular expression. For example, the following correctly parses a number:

```
number = [ "literal_re", "[0-9]+" ];
```

More information on perl regular expressions can be found in any good perl book or on the web.

Blocks

Now we shall look at adding brackets to our notation. This will give our calculator the ability to work out expressions like ‘(1+2)*3’. We have seen how to use the "prefix" and "suffix" operations, so we can use these to parse brackets:

```
expr5 = [ "prefix", "(", "expr6", [ "fail", "term" ] ];

expr6 = [ "suffix", ")", "expr" ];
```

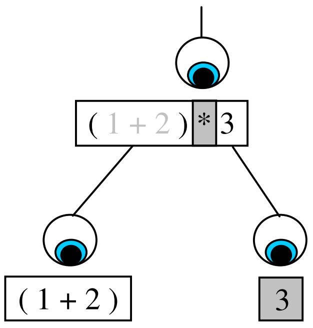
This gives us a bit of a problem. Think about the input ‘(1+2)*3’. The first agent will use the pivot operation on the ‘+’, leaving two substrings ‘(1’ and ‘2)*3’. Instead, we really want the parser to pivot on the ‘*’ and break the input into the two smaller expressions ‘(1+2)’ and ‘3’. The parser needs to be sensitive with our brackets.

This is where blocks are a very useful feature of the Agent-oriented Parser. We can define a block and instruct agents to ignore that block. To define a block:

```
bras = [ ["(", ")"], ["bras"] ];

addblocks("bras");
```

The first statement is the block definition. The first item of the list is a pair of strings, the first being the starting string of the block and the second the end string. The second item is a list containing names of blocks that may be contained within the block. The second statement adds the block definition to the environment.

Now for a particular rule we can specify it to ignore blocks. For our calculator notation, we want to ignore any strings between brackets when the agent is looking for an arithmetic operator (+,-,*,/). We add an ignore clause to our rule:



```
expr = [ "pivot", "+", "expr", "expr" ,

         [ "ignore", ["bras"] ],

         [ "fail", "expr2" ] ];
```

This behaviour is demonstrated in the diagram (right). The string within the brackets is ‘greyed-out’ because it is ignored. Hence the most important string to be observed is the multiplication sign (*) and a pivot is made.

It is important to remember that when an agent ‘ignores’ a block it is not removing that block from the input. It is perhaps better described as preserving the block. The agent simply preserves the contents of the block and leaves it for another agent to parse.

Scripting

A parser that either accepts or rejects an input is of little use unless it produces some output. Our simple calculator needs some way of outputting the result of an expression. This is achieved with agent actions. If an agent does not fail to match the input then it can optionally perform some actions. Each action can be performed before or after the actions of the agent's children.

The format for including agent actions in a rule definition is similar to the other optional components of a rule. Here we modify our "term" rule by adding an action that prints out some random comment:

```
term =
  [ "literal_re", "[0-9]+",
    [ "action",
      [ "now", "writeln(\"somerandomcomment\");" ] ] ];
```

The third item in the list above is the action declaration. This sublist begins with the "action" tag to recognise it from the other optional tags. The items following the head tag are the commands to execute. Each command is a list containing only 2 items, the first being either "now" or "later" depending on whether the command will be executed before or after the child agents. The second part of the command is the command string which is typically some eden code to be executed.

An agent has some data associated with it that can used in its actions. Each agent also has a unique variable associated with it. This agent data can be substituted into the command string using the following:

```
$i = the input string to the agent
$j = the name of a variable containing the input string
$t = the token/string that was matched by the agent
$v = the variable name that belongs to the agent
$s1 = the first substring of the input
$s2 = the second substring of the input (and so on for 3rd,
      4th, ..)
```

```
$p1 = the variable name of the first child agent (parameter 1)

$p2 = the variable name of the second child agent
```

Now we can make our "term" rule more useful by adding an action that stores the term value in the agent variable:

```
term =

  [ "literal_re", "[0-9]+",

    [ "action",

      [ "now", "$v = $t;" ] ] ];
```

We would then add actions to our other rules. The "expr" rule can do the addition of the two sub-expressions:

```
expr =

  [ "pivot", "+", [ "expr", " expr" ],

    [ "action",

      [ "later", "$v = $p1 + $p2;" ] ],

    [ "fail", "term" ] ];
```

Take a look at the final calculator notation at the end of the document for more examples of agent actions.

Note: The dollar sign is a special character in the command string. If you want to print a single dollar sign ($) in your command string then you must follow it by another dollar sign ("$$" will produce a single dollar in the command string).

The original version of the Agent-oriented Parser had a different method for writing scripts, using the "script" tag. Although the parser will still accept these scripts, it is recommended you use the "action" notation. For more information on the "script" tag, refer to Chris Brown's third year project [Bro01].

Installing notations

Now that we are happy with our calculator notation, it is probably a good idea to make it more accessible. We can install new notations into the Eden environment, which can then be used in scripts as you would existing notations (e.g. %eden, %donald, %scout). In tkeden this will add a radio button for our new notation to the environment.

First we must create an initialisation rule:

```
calc_init = [ "\n", "calc", [] ];
```

The first item in the list is the string to split the input on. For our calculator notation we would like to separate each command by the end-of-line character (\n). For other notations you may wish to split the input on other characters (e.g. semi-colon for C/Java/Eden). The second item is the starting rule. The third item is a list of blocks to ignore in the splitting procedure.

To install the notation in the environment:

```
installAOP("%mynotation", "calc_init");
```

Notations must begin with a percent (%) character.

You can now switch to the new notation by typing `%mynotation`. Do not forget to switch back to `%eden` for Eden code!

The final calculator notation

```
calc_init =

  [ "\n", "calc", [] ];


calc =

  [ "prefix", "", "calc_expr",

    [ "action",

      [ "later", "writeln('=',$p1);" ] ],

    [ "fail", "calc_err" ] ];


calc_expr =

  [ "pivot", "+", [ "calc_expr", "calc_expr" ],
    [ "ignore", ["bras"] ],

    [ "action",

      [ "now", "$v is $p1 + $p2;" ] ],

    [ "fail", "calc_expr2" ] ];


calc_expr2 =

  [ "pivot", "-", [ "calc_expr", "calc_expr" ],

    [ "ignore", ["bras"] ],

    [ "action",

      [ "now", "$v is $p1 - $p2;" ] ],

    [ "fail", "calc_expr3" ] ];


calc_expr3 =

  [ "pivot", "*", [ "calc_expr", "calc_expr" ],

    [ "ignore", ["bras"] ],

    [ "action",

      [ "now", "$v is $p1 * $p2;" ] ],

    [ "fail", "calc_expr4" ] ];


calc_expr4 =

  [ "pivot", "/", [ "calc_expr", "calc_expr" ],

    [ "ignore", ["bras"] ],

    [ "action",

      [ "now", "$v is $p1 / $p2;" ] ],

    [ "fail", "calc_expr5" ] ];
```

```
calc_expr5 =
  [ "prefix", "(", "calc_expr6",
    [ "action",
      [ "now", "$v is $p1;" ] ],
    [ "fail", "calc_term" ] ];


calc_expr6 =
  [ "suffix", ")", "calc_expr",
    [ "action",
      [ "now", "$v is $p1;" ] ],
    [ "fail", "calc_err" ] ];


calc_term =
  [ "literal_re", "[0-9]+",
    [ "action",
      [ "now", "$v = $t;" ] ],
    [ "fail", "calc_err" ] ];


calc_err =
  [ "read_all", [],
    [ "action",
      [ "now", "writeln(\"calc: syntax error\");" ] ] ];


installAOP("%calc", "calc_init");
```

Extensions

If you want to experiment with this notation, then here are a few ideas of how to extend it:

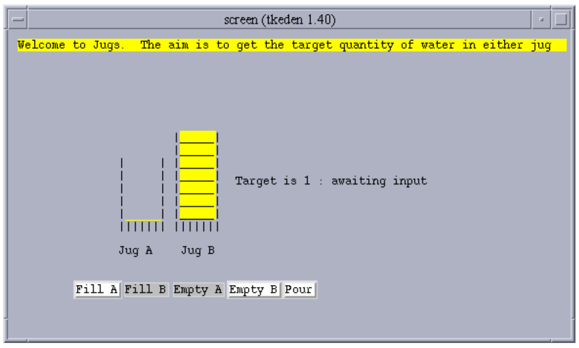
- Add some common constants like ‘pi’ and ‘e’.
- Introduce power and square root functions.
- Add memory capabilities like you would normally find on a calculator (e.g. M+, MR, etc).

Appendix C – Glossary of models used in the thesis

This appendix contains brief descriptions of the EM models used as case studies in this thesis. For each model, we give a reference to its location in the EM model repository [EMRep], a short description of the model and a screenshot of it in use. For further details on individual models, consult the documentation files provided with each model in the repository.

The jugs model – [EMRep, jugsBeynon1988] and [EMRep, jugsPavelin2002]

This is a model of a simple educational game first developed for the BBC computer in the 1980's by the Chiltern Advisory Unit. The game revolves around trying to measure a specified amount of water where there are two jugs of known quantities that have no markings on them. A set of basic operations is available on the interface to empty a jug, fill a jug, or pour water between the jugs.

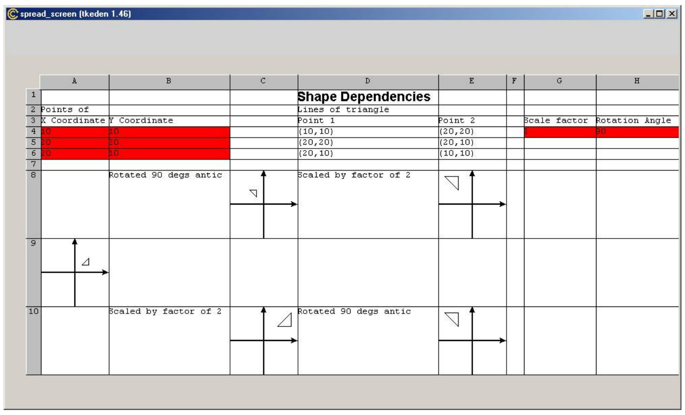


Welcome to Jugs. The aim is to get the target quantity of water in either jug

Target is 1 : awaiting input

The spreadsheet model – [EMRep, spreadsheetRoe2002]

This spreadsheet created using TkEden illustrates connections between spreadsheets and modelling. The model can replicate the essential functionality of conventional spreadsheets and can show how the

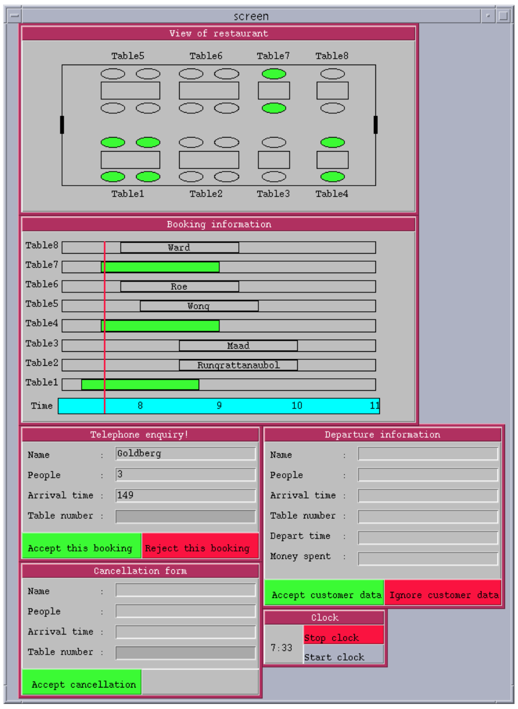


	A	B	C	D	E	F	G	H
1				Shape Dependencies				
2	Points of			Lines of triangle				
3	X Coordinate	Y Coordinate		Point 1	Point 2		Scale factor	Rotation Angle
4	10	10		(10,10)	(20,20)		2	90
5	20	20		(20,20)	(20,10)			
6	20	10		(20,10)	(10,10)			
7								
8		Rotated 90 degs antic		Scaled by factor of 2				
9								
10		Scaled by factor of 2		Rotated 90 degs antic				

generalised notion of dependency in EM allows the spreadsheet model to support a wider variety of types. It also illustrates agent actions in a spreadsheet.

The restaurant model – [EMRep, restaurantRoe2000]

This case study illustrates how a model can be constructed to investigate restaurant management. The user of the model can play through fictional scenarios of bookings in the restaurant to gain experience of how to best allocate tables in order to maximise profit. The model contains a seating plan for a fictional restaurant, a timetable booking sheet and a set of forms to enter data in. Scenarios can either be created manually or a random sequence of events can be generated to simulate the activity on an evening.



Booking information	
Table8	Ward
Table7	
Table6	Roe
Table5	Wong
Table4	
Table3	Maad
Table2	Rungrattanaubol
Table1	

Telephone enquiry!

Name :	Goldberg
People :	3
Arrival time :	149
Table number :	

Accept this booking | Reject this booking

Cancellation form

Name :	
People :	
Arrival time :	
Table number :	

Accept cancellation

Departure information

Name :	
People :	
Arrival time :	
Table number :	
Depart time :	
Money spent :	

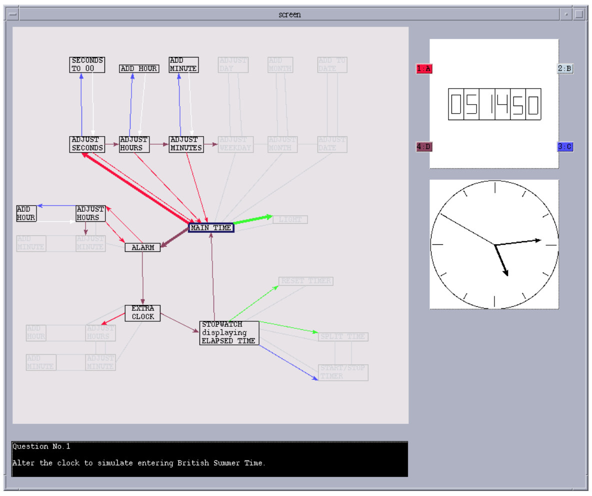
Accept customer data | Ignore customer data

Clock

7:33 | Stop clock | Start clock

The digital watch – [EMRep, digitalwatchRoe2001]

The digital watch model has been developed by a number of different people over a period of eight years. It consists of a digital watch display with a number of buttons to activate its functionality, an analogue clock display and a graphical depiction of all the states that the watch can be in. States emerge when they have been visited for the first time.

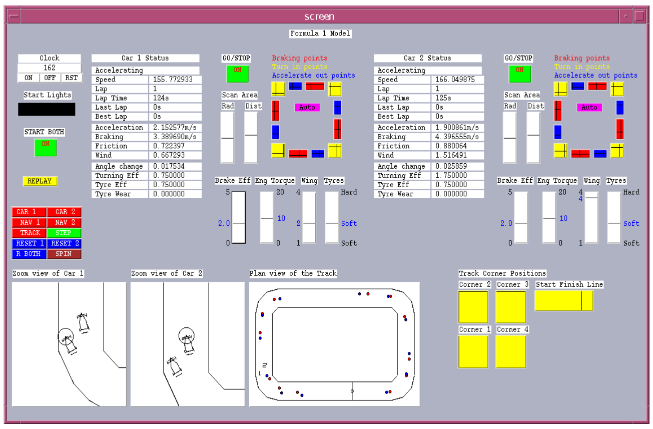


Question No.1

Alter the clock to simulate entering British Summer Time.

The racing cars model – [EMRep, racingGardner1999]

This model can be used to explore the setup of racing cars in order to minimise lap times around a track. It is layered so that learners are exposed to more functionality at each successive layer. The final layer of the model is shown in the screenshot, and contains two fully customisable cars that race against each other on a partially customisable track.



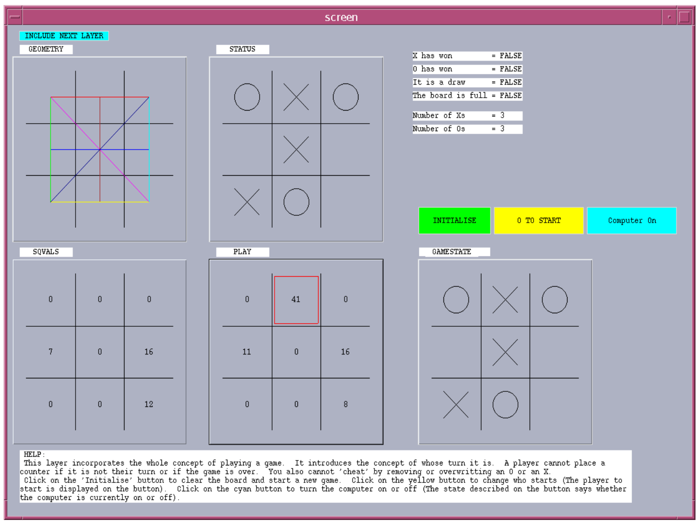
Clock: 162

Car 1 Status	
Accelerating	
Speed	155.772933
Lap	1
Lap Time	124s
Last Lap	0s
Best Lap	0s
Acceleration	2.152577m/s
Braking	3.389690m/s
Friction	0.722397
Wind	0.667293
Angle change	0.017534
Turning Eff	0.750000
Tyre Eff	0.750000
Tyre Wear	0.000000

Car 2 Status	
Accelerating	
Speed	166.049875
Lap	1
Lap Time	125s
Last Lap	0s
Best Lap	0s
Acceleration	1.900861m/s
Braking	4.396555m/s
Friction	0.880064
Wind	1.516491
Angle change	0.025859
Turning Eff	1.750000
Tyre Eff	0.750000
Tyre Wear	0.000000

The OXO model – [EMRep, oxoJoy1994], [EMRep, oxoGardner1999] and [EMRep, 3doxoRoe2001]

This is a layered model that introduces different concepts of noughts-and-crosses at each layer. The layers introduce the board and its geometry, the pieces to be placed on the board, the rules by which the pieces can be placed on the board, and strategy considerations for a computer player. At each layer there is no prescription about future layers that are to be added.



X has won = FALSE
0 has won = FALSE
It is a draw = FALSE
The board is full = FALSE

Number of Xs = 3
Number of 0s = 3

SQVALS

0	0	0
7	0	16
0	0	12

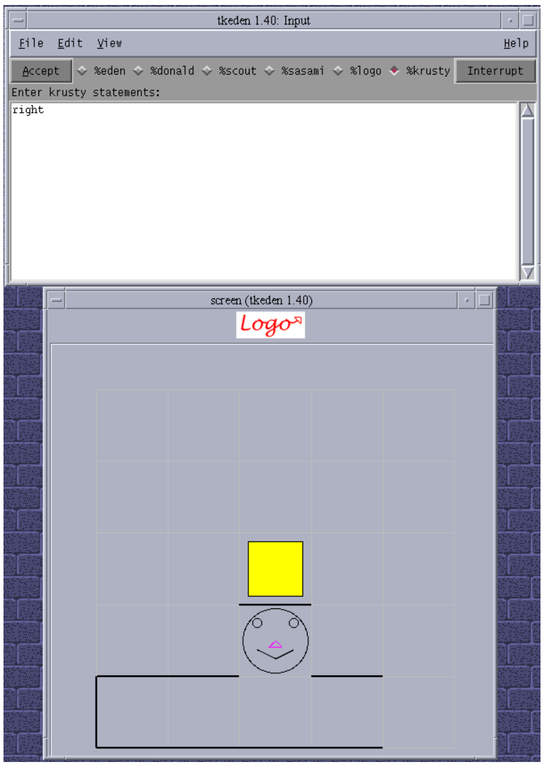
PLAY

0	41	0
11	0	16
0	0	8

HELP:
This layer incorporates the whole concept of playing a game. It introduces the concept of whose turn it is. A player cannot place a counter if it is not their turn or if the game is over. You also cannot 'cheat' by removing or overwritting an 0 or an X.
Click on the 'Initialise' button to clear the board and start a new game. Click on the yellow button to change who starts (The player to start is displayed on the button). Click on the cyan button to turn the computer on or off (The state described on the button says whether the computer is currently on or off).

The clown-and-maze model – [EMRep, krustyRoe2002]

The clown-and-maze model illustrates how languages for interaction can be interactively extended from very simple languages. In this model, the initial language contains just four simple directional commands and gradually the functionality of a Logo style language is added. The language for interaction can be interactively altered whilst the model is running. The aim of the underlying model is to direct a clown around a maze to find treasure.

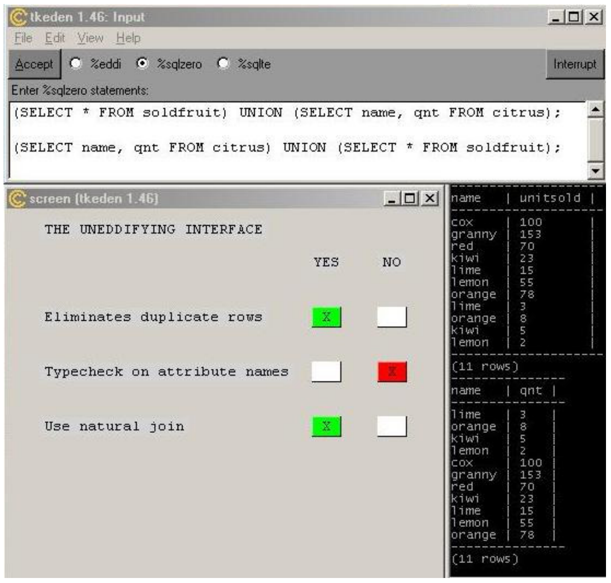


Enter krusty statements: [✓ %krusty]

```
right
```

The SQL-EDDI model – [EMRep, sqleddiWard2003]

The SQL-EDDI environment has been used on the 2nd year Introduction to Database Systems module at the University of Warwick to explore the relationship between relational algebra and relational query languages. It comprises an interpreter for a subset of SQL and a relational algebra language ("EDDI"). 'The Uneddifying Interface' is used to control the way in which queries are evaluated and illustrate flaws in the design of SQL.



Enter %sqlzero statements: [✓ %sqlzero]

```
(SELECT * FROM soldfruit) UNION (SELECT name, qnt FROM citrus);

(SELECT name, qnt FROM citrus) UNION (SELECT * FROM soldfruit);
```

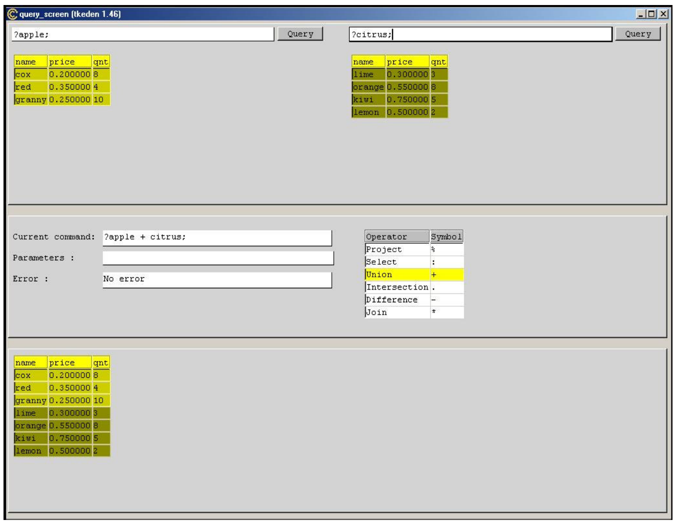
THE UNEDDIFYING INTERFACE

	YES	NO
Eliminates duplicate rows	🟩 X	
Typecheck on attribute names		🟥 X
Use natural join	🟩 X	

```
------------------------
name    | unitsold |
------------------------
cox     | 100      |
granny  | 153      |
red     | 70       |
kiwi    | 23       |
lime    | 15       |
lemon   | 55       |
orange  | 78       |
lime    | 3        |
orange  | 8        |
kiwi    | 5        |
lemon   | 2        |
------------------------
(11 rows)
----------------
name    | qnt |
----------------
lime    | 3   |
orange  | 8   |
kiwi    | 5   |
lemon   | 2   |
cox     | 100 |
granny  | 153 |
red     | 70  |
kiwi    | 23  |
lime    | 15  |
lemon   | 55  |
orange  | 78  |
----------------
(11 rows)
```

The RAT – [EMRep, ratRoe2003]

The Relational Algebra Tutor was developed to show how the basic operations of relational algebra produce output tables from one or two input tables. The operations are colour coded so that the learner can see directly how an output table is created. The syntax of the resultant relational algebra query in the EDDI language is displayed.



?apple;

name	price	qnt
cox	0.200000	8
red	0.350000	4
granny	0.250000	10

?citrus;

name	price	qnt
lime	0.300000	3
orange	0.550000	8
kiwi	0.750000	5
lemon	0.500000	2

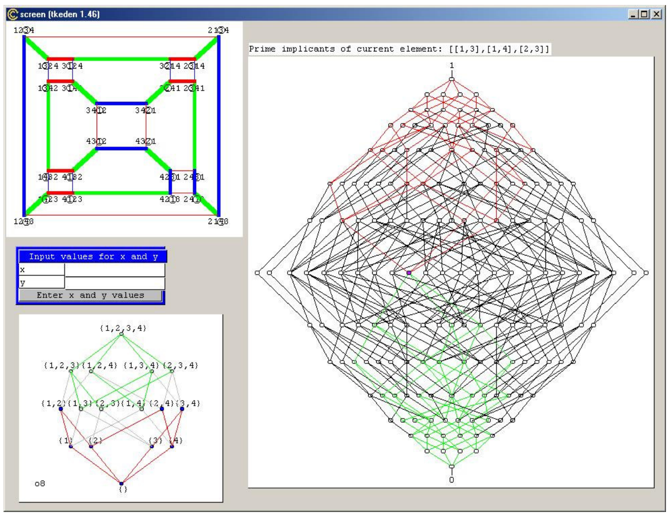
Current command: ?apple + citrus;
Parameters :
Error : No error

Operator	Symbol
Project	%
Select	:
Union	+
Intersection	.
Difference	-
Join	*

name	price	qnt
cox	0.200000	8
red	0.350000	4
granny	0.250000	10
lime	0.300000	3
orange	0.550000	8
kiwi	0.750000	5
lemon	0.500000	2

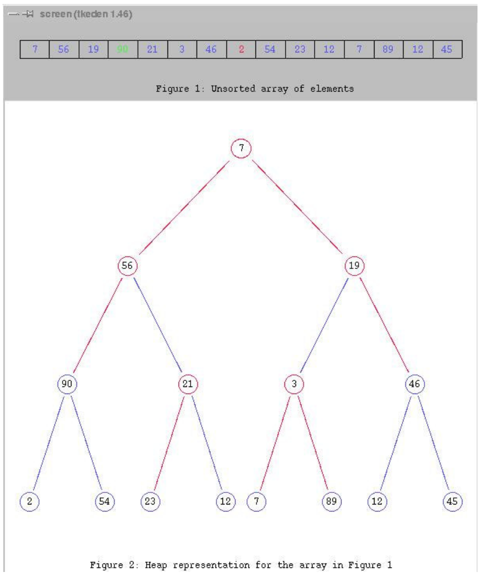
The MBF4 model – [EMRep, mbf4Beynon2003]

The MBF4 model illustrates how several independently developed models can be integrated into a single model using dependency. The model was developed to explore connections between different realisations of an abstract area of mathematics surrounding lattice theory.



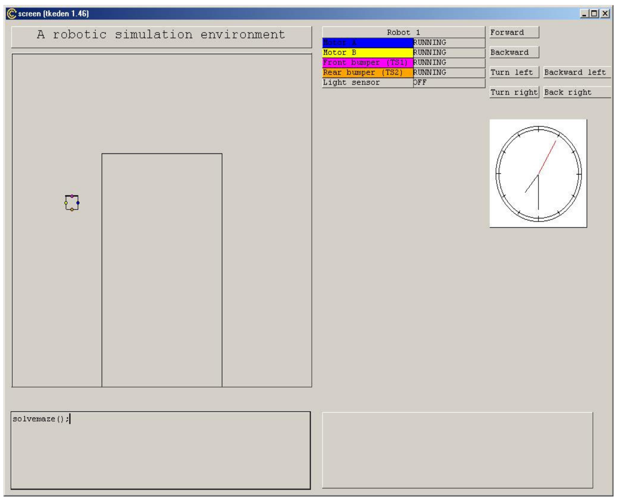
The heapsort model – [EMRep, heapsortBeynon1998]

The heapsort model is an example of a partially built model to which a learner can add small fragments of script to embellish the basic model. The initial model consists of a heap structure that a learner can manipulate to gain understanding of the essential nature of a heap. Agent actions can be added to the model to introduce the basic transitions of heapsort that are eventually directed at automation of the heapsort algorithm. By building up the model gradually, a learner can introduce new definitions as their competency increases.


screen (tkeden 1.46)
7 56 19 90 21 3 46 2 54 23 12 7 89 12 45
Figure 1: Unsorted array of elements
7
56
19
90
21
3
46
2
54
23
12
7
89
12
45
Figure 2: Heap representation for the array in Figure 1

The RSE – [EMRep, rseRoe2003]

The robot simulation environment is a prototype for understanding programming of LEGO Mindstorms robots. The environment is an exploratory laboratory where learners can directly manipulate robots in order to understand how the movement of the motors and the state of the sensors is related to the real-world behaviour of the robot.



Robot 1	
Motor A	RUNNING
Motor B	RUNNING
Front bumper (TS1)	RUNNING
Rear bumper (TS2)	RUNNING
Light sensor	OFF

```
solvemaze();
```

Appendix D – Example model building interactions in EM

The following extract is taken from the paper 'Liberating the Computer Arts' by Meurig Beynon [Bey01]. This extract describes model building in EM with reference to the construction of a simple analogue clock.

4.2. Principles and Tools of EM

The principles and tools of EM will be briefly described and illustrated with reference to a simple but extended example. This focuses on model-building activity surrounding an imaginary analogue clock.

The primary focus for representation in EM, as in art – and in contrast to computer programs that are targeted at behaviour, is on situation. In the initial situation in which we observe the modelling activity, the model consists of the outline of the clock. This is defined by a family of definitions of variables (a definitive script) in a special-purpose notation (a definitive notation) for line-drawing – DoNaLD.

```
%donald

viewport CLOCK


openshape clock


within clock {

    real sixthpi

    line eleven, ten, nine, eight, seven, six, five, …, one

    line noon
    point centre

    real radius

    circle edge


    sixthpi = 0.523599

    radius = 150.0

    eleven = rot(noon, centre, -11 * sixthpi)

    ...

}
```

The variables in this script represent observables in the clock: the rim of the face, represented by the circle `clock/edge`, its centre `clock/centre` and the divisions `eleven, ten, nine,` ... etc that indicate the hours.

The artefact that is defined in this fashion is depicted on a screen, as in Figure 1.1:

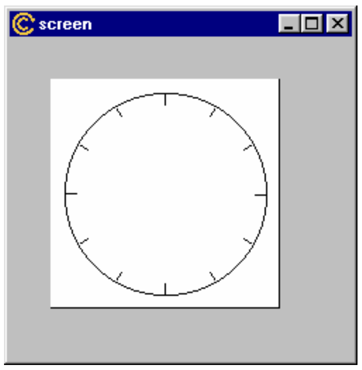


Figure 1.1: The clock face

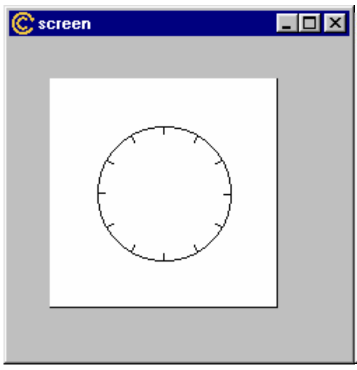


Figure 1.2: The scaled face

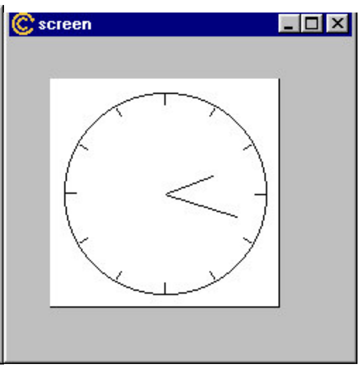


Figure 1.3: With time set

The definitions in the script establish dependencies between the variables similar to those in a spreadsheet. Interaction with the script takes place in an environment in which the values of variables are always open to redefinition. For instance, the redefinition

```
clock/radius = 100.0
```

has the effect of making the clock smaller, simultaneously changing all the positions of the divisions and the rim of the clock. The display also depends directly upon the values of variables in the script, and is simultaneously updated (see Figure 1.2).

The hands of the clock are not yet displayed, but there are already variables and definitions in the script that refer to them:

```
within clock {
  line minHand, hourHand
  real minAngle, hourAngle
  real size_minHand, size_hourHand
  int t
  size_minHand, size_hourHand = 0.75, 0.5
  minAngle = (pi div 2.0) - float (t mod 60) * (pi div 30.0)
  hourAngle = (pi div 2.0) - float (t mod 720) * (pi div 360.0)
  minHand = [centre + {size_minHand*radius @ minAngle}, centre]
  hourHand = [centre + {size_hourHand*radius @ hourAngle}, centre]
  centre = {200, 200}
  ....
}
```

They are not currently displayed because the value of the integer `t`, which represents the current time, has yet to be determined. Assigning different values to `t` readily establishes that `t` can be interpreted as the time elapsed in minutes from midnight, so that (for instance) assigning `t` to 138 sets the clock to time 2.18 (see Figure 1.3.).

The modeller's interaction with the clock script is not directed towards any particular goal or constrained by a preconceived interpretation. The modeller views interaction with the artefact from the perspectives of many different human agents, shifting perspective arbitrarily, much in the way that an artist saturates their imagination through attention to the emerging work of art. For instance, in developing a play a playwright might reflect upon a particular situation from the viewpoint of the fictitious characters in the play, the actors, the audience or the producer. The simultaneous consideration of all these viewpoints is not associated with a separation of concerns, but with a dwelling in the situation so as to draw out all its possibilities and enrich the experience of the author in the writing and the audience in the appreciation of the work. This holistic approach is vital to the activity, and is complemented by an openness and responsiveness to what is encountered that characterises creative thought.

The act of setting the time on the clock supplies a modest illustration. It might be that the clock is to be sold, and the hands placed in the most aesthetically pleasing configuration. It might be that the user is setting the clock to the current time. The time on the clock potentially represents an aspect of the clock that is beyond user control. The script is intended to reflect all these possible interpretations, and support them in so far as they can co-exist in the modeller's imagination. The novelty in this approach lies in the ontology of the model: there is no prescribed interpretation, only certain interpretations that may acquire particular significance and permanence as the modeller's imagination shapes the model and the referent.

The description of EM activity is framed in terms of 'how the modeller construes the situation'. Observables, agency and dependency are the key concepts used to express construals. Where there is a fixed external physical referent, construal is concerned with how the observations of the referent can be explained with reference to agency, dependency and observables. It is also appropriate to regard the relationship in which the modeller chooses to stand to the referent as a form of construal, where the emphasis is placed upon the agency of the observer rather than agency that operates within the referent. Interaction with the script can be associated with elaborating a construal of either kind. The parametrisation of the clock face in terms of `noon` and `radius` is concerned with construal from the modeller's perspective. As a simple illustration of how construal applies to the clock mechanism itself, the following revised definition for `hourAngle` establishes the dependency between minute and hour hand that is typically present in a mechanical clock:

```
within clock {

      minAngle = (pi div 2.0) - float (t mod 720) * (pi div 30.0)

      hourAngle = (pi div 2.0)  - ((pi div 2.0) - minAngle) div 12.0

      ...

}
```

Through this redefinition, `hourAngle` depends upon `minAngle`, and is no longer directly defined in terms of the time t. Note also that the definition of `minAngle` has also been modified (so that it records the reflects the number of minutes elapsed over a 12 hour period rather than a single hour) so that it delivers the correct result for `hourAngle`. This can be interpreted as reflecting whether or not we are taking the state of the internal mechanism of the clock into account in observing the position of the minute hand. The relationship between these two scripts illustrates how the removal of dependency can be associated with optimisation and information loss.

The practical techniques that the modeller can use to support construal include the use of definitions to express dependency and the introduction of triggered procedures to represent agent actions. The underlying framework is supplied by the EDEN interpreter, which serves both as an "evaluator of definitive notations" (in particular for DONALD) and as a hybrid definitive/procedural environment that interfaces with the user and the computer. Through EDEN it is possible to increment the variable `clock/t` repeatedly for instance, so as to simulate the clock in operation. The speed at which it is appropriate to carry out this update is at the discretion of the modeller: it may be useful to observe the hands in slow motion, as in the construction of the rotating clock described below, to run it as fast as possible to obtain an overall impression of the clock behaviour, or to synchronise the update of variable `clock/t` with the system clock.

The above discussion illustrates many of the distinctive features of EM in a simple setting. The most significant issue is that the context is such that the term modelling is not entirely apposite (so obscure is the referent and the goal of the modeller): there are many roles for agency, the conventions for observation and interpretation are fluid, and modelling activity is as much concerned with exploring what the referent is as with representing it.

4.3. The SIN principle in EM

The messiness of our real engagement with the world is at odds with the systematic models of behaviour to which a science aspires. In building artefacts that can support this engagement, the idea of making things easy for the user is suspect. It is appropriate to eliminate unnecessary frustration, but not to suppose that all frustration can be eliminated (cf. Donofrio's [IBM Vice-President] "what we want, when we want it, where we want it"), or even that that would be a desirable goal. Art works often both with and against the tools and the medium, and this is not something that can be designed away. One aspect of the artist's skill is to overcome the limitations of the instrument: "a bad workman blames his tools".

The evidence is that in certain respects the tools of EM are not easy to use. Even after making allowance for some obvious flaws in the tools – and acknowledging that a bad designer blames the workman – there is an essential reason for difficulty in use. It is quite usual in conventional programming to achieve an end without explicitly considering what assumptions have been made in order to achieve it. In EM, there is no choice but to engage with the experience that should inform our constructions. This activity is expensive in terms of human time and effort – but there is no substitute for it. The potential advantage of the EM approach is that, when we subsequently identify problems, the model itself can help us to access the knowledge that informed our original solution, and that is required to improve this solution.

An actual illustration drawn from developing a rotating clock model is useful at this point. I wish to highlight the noon position on the clock by marking it with a longer line segment. I choose to do this because of the limitations of the tools: the attributes of a line drawing are not preserved if I create a new image of it by rotation. I first think of refining the line `noon`, but realise that this will affect all the other divisions by dependency. This means that I shall introduce a new line `noon2` to mark the noon division. My idea is to derive `noon2` by elongating the line `noon`. This I can do by addressing its endpoints `noon.1` and `noon.2`. My first attempt is:

```
within clock {
        line noon2
        noon2 = [noon.1, noon2*2]
        ...
}
```

This is identified as a cyclic definition, since `noon2` appears on the right hand side of the definition of `noon2` – a mistake precipitated by my own choice of notation. I correct the redefinition of `noon2` thus:

```
noon2 = [noon.1, noon.2*2]
```

This is a conceptual mistake – it results in a spike at `noon`, not a longer line – I am forgetting that the elongation has to be along the direction of the line `noon`. I next try:

```
noon2 = [noon.2 + (noon.1 - noon.2)*2];
```

This a type checking mistake – I am getting confused about what sub-expressions are lines and what are points. To correct this, I enter:

```
noon2 = [noon.2 + (noon.1 - noon.2)*2, noon.2]
```

This is a mistake because the endpoints `noon.1` and `noon.1` are not the way round that I thought they were. Only then do I get what I wanted:

```
noon2 = [noon.1, noon.1+ (noon.2 - noon.1)*2]
```

The interpreter itself poses its own syntactic challenges to correct input, and the steps detailed above were further complicated by such vagaries. Pencil and paper also played a role in supporting his interaction. On several counts, EM offers poor quality end-user interaction, but it is unusual in that it supports a degree of engagement even in error and misconception.

The experimental form of my interaction points to a significant danger: that EM encourages sloppiness in thought and practice. That said, I know that the kind of activity exposed above is quite characteristic of my inner thought processes, and (modulo the vagaries of the tools) that it is harder to trace this on paper or in my head. The end result is also much more satisfactory – not only do I derive the correct answer, but I construct an environment in which my mistakes and misconceptions have been captured and recorded to an extent that is otherwise problematic.

As highlighted in this example, 'making mistakes' is an essential part of employing EM. For the traditional programmer, this is a difficult concept: the most significant mistakes that the skilled programmer makes are in the early stages of design, and hopefully never reach the implementation. Though it can be frustrating and embarrassing to follow through the experimental phases of design in EM interactively – and whilst it is tempting to focus on flaws in the notations, the interface or the interpreter – there is some virtue in exposing our imperfect thought processes.

4.4. Illustrating the SIN principles of representation in EM

This section develops the theme of the simple clock model to illustrate how representation operates in EM. The significance of using EM in representation is best understood by considering the continuity in the cognitive activity that accompanies the modelling. When a conventional program is executing, there are at least two aspects of its state that are relevant. There is the computational state, with which the user may or may not be interacting, which – in so far as it is intended to be interpreted by the user – is meaningful in terms of the application of the program. There is also –

at a meta-level – the state of the program code itself, which is not typically known to the user.

When making a redefinition in a script, the modeller can be changing the associated state in what conventionally would be regarded as affecting both of these aspects. On the one hand, the state that is visible to the user may be changed. On the other, the underlying pattern of dependency maintenance ("part of the program code") may also be changed. The aspiration for EM, to some considerable degree supported even by our current tools, is for it to be possible to change the program code without disrupting that part of the state with which the user is engaging. An analogy can be made between a conventional programming paradigm and the way in which a traveller might use a vehicle built by a mechanic at a workshop, travel about in it to discover its limitations, then return to the workshop so that it can be modified to overcome these. The aspiration for EM is that the mechanic and his workshop can journey with the traveller, to effect modifications in the context where they are needed, potentially with more first-hand appreciation of the requirement.

A few examples will serve to illustrate how EM helps to address issues concerned with situation, ignorance and nonsense.

- Representing situation

The representation of situation in EM can be illustrated in many ways. For instance, to set the clock to Japanese time:

```
%eden

uk_time is tnsecs / 60;

/* tnsecs = the number of seconds that have elapsed since a fix date
*/

japan_time is uk_time + 480;

_clock_t is japan_time;
```

To represent a broken clock, in which the minute hand hangs loose:

```
%donald

clock/minAngle = - pi div 2
```

It is also possible to take account of observables present in the situation, but previously unrecorded. For example, to add a secondhand that is coloured red (see Figure 3):

```
%donald

within clock {

      line secHand

      real secAngle, size_secHand

      secHand = [centre + {size_secHand*radius @ secAngle}, centre]

      size_secHand = 0.8

}

%eden

sec_mod_60 is tnsecs % 60;

A_clock_secHand = "color=red";
```

- Representing ignorance

The primary respect in which EM deals with missing knowledge is through supporting variables whose value is as yet undefined. The graceful handling of the unspecified time on the clock discussed above is a simple illustration.

Another kind of ignorance is that associated with exploratory design, where something is known only after it is constructed and recognised in interaction. As a simple illustration, the modeller can act in the role of a clock designer via the redefinition:

```
within clock {
      circle inner_edge
      real width_edge
      inner_edge = circle(centre, radius - width_edge)
      width_edge = 20.0
      ...
}
```

Such a redefinition is here to be construed as changing the clock itself.

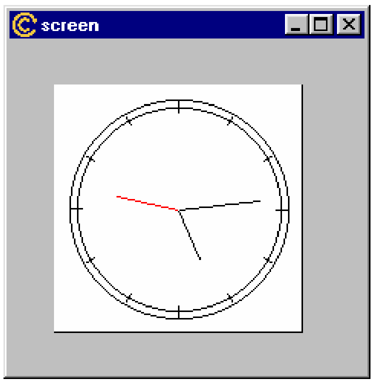


Figure 2: Hand and Rim added

Further experimentation with values of `width_edge` elicits tacit knowledge about what is an acceptable design through interaction.

Conventional programming is knowledge-driven, in that a program is designed with a specific functionality in mind. Being able to make use of a model even though we are ignorant about how we might wish to use it is possible to the extent that we have an effective construal. The rotating clock, in which the clock pivots freely about its centre according to the moments associated with its hands, was conceived as an opportunistic extension of the clock model. This construction illustrates the way in which automatic agency, such as here represents the clock rotating into equilibrium as time passes, is developed from – or as if from – a pattern of interaction by the modeller. For instance, the modeller might compute the moments of the clock with minute hand and hour hand pointing vertically downwards, then when the rotation of the clock lies halfway between these positions, and proceed by binary search to locate the rotation that makes the moment of the clock zero. This can be represented by using the pattern of observation:

```
%eden

hA = _hrAngle;

mA = _mnAngle;

momentH is moment(_hrAngle, _mnAngle, hA);

momentHM is moment(_hrAngle, _mnAngle, (hA+mA)/2.0);

momentM is moment(_hrAngle, _mnAngle, mA);
```

– where `_hrAngle` and `_mnAngle` refer to the current positions of the hour and minute hands in the normal sense, and the function `moment()` returns the moment of the clock about the angle specified by its third argument – and assigning `hA` or `mA` to `(hA+mA)/2` according to the sign of `momentHM`. This binary search can then be performed automatically by an agent that responds to the changing time. This illustrates how EM can be used as a way of specifying the functions that maintain dependencies themselves.

The use of this model to explore the dynamics of the rotating clock is another more familiar sense in which EM is concerned with the discovery of knowledge rather than its exploitation.

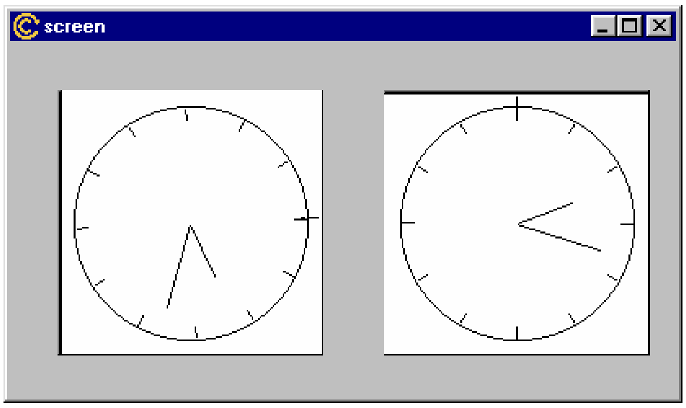


Figure 3: The rotating clock

- Representing nonsense

A much-neglected concern in our representation of the world is the way in which our minds impose relationships upon possible situations and events. We are accustomed all the while to organise our experience according to the degree to which it is familiar and 'makes sense'. This leads us to say "I don't think that's possible", or "if that can happen, then that is also plausible". Though nonsense suggests the antithesis of sense, it in fact refers to what is – in this mind-space – near enough to sense to blend with it in some respects. Perhaps the most important feature of the EM representation of a situation by a script is that it establishes such relationships: namely, nearness as assessed by the kind of redefinition that is required to transform one script to another, and by the kind of agency that would be involved. The extent to which these relationships match those that we encounter in the world reflects the quality of our construal, as determined by the observables, dependency and agency we identify. The clock study is a useful source of examples.

It is easy to modify the clock so that the length of the second hand depends on the time:

```
_clock_size_secHand is (float(sec_mod_60)/60.0) *
_clock_size_minHand;
```

In the days of the mechanical clock, this would have been implausible if not nonsensical, but it would seem unremarkable on a computer desktop. There are simple redefinitions to create a mirror image clock:

```
%scout

window clockwin = {

    type: DONALD

    box: [clockwinNW, clockwinSE]

    pict: "CLOCK"

    xmin: 30

    ymin: 370

    xmax: 370

    ymax: 30

    bgcolor: "white"

    border: 1

};
```

or an the upside-down clock:

```
%donald

clock/radius = -100.0
```

These examples highlight the role that agency of the observer plays in discriminating sense from nonsense: the conventions by which we read the time are in principle so arbitrary. The rotating clock with nothing to distinguish noon from other divisions would be more absurd as a timepiece.

Another extension of the basic clock model serves to illustrate how visual art exploits both the sense of space and 'the space of sense'.

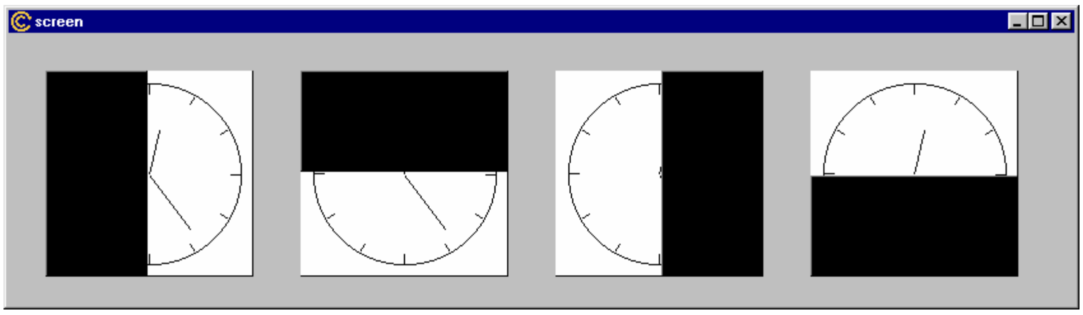


Figure 4: A model loosely based upon Richard Wentworth's The Warwick Dials (1999)

This model is loosely based on a construction on display in the Warwick Arts Centre. It is made by a simple extension of the original model in which the line drawing of the clock is displayed in several windows, each partially occluded by a blank window whose size can be altered by redefining the variable `blankedge`.

```
%scout

integer blankedge = 83;


window blank1 = {

    type: DONALD

    box: [clockwinNW, clockwinSE - {blankedge*clockwinScale, 0}]

    bgcolor: "black"

    border: 1

};

...


display     screen     =     <blank1/clockwin/blank2/clockwin2/blank3/
clockwin3/blank4/clockwin4>;
```

In the space of sense, there is a distinction between nonsense and meaninglessness. In EM, this can be explored by transforming artefacts through random redefinition of variables. The arbiter in matters of sense is the observer, who may or may not be able to connect in any way with the experience offered by the transformation of an artefact. The redefinition of variables that correspond to observables beyond the control of any recognised agent, and redefinitions that subvert our physical intuitions about the permanence and reliability of objects are some of the most effective in destroying the semantic relation, as in the corrupted clock in Figure 5:

```
within clock {

      sixthpi = 1.0

      minHand = [centre + {0.75*radius @ minAngle}, hourHand.1]

      ...

}
```

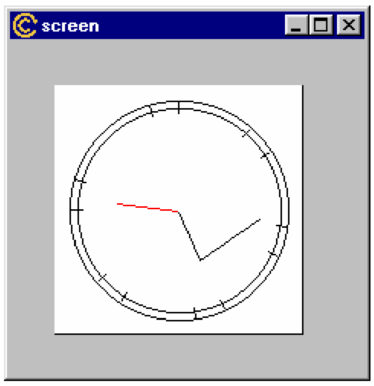


Figure 5: The corrupted clock