

# CS1412: Artificial Intelligence Fundamentals

Laboratory Exercises 2004-2005

September 17, 2004

## 1 Welcome to Hector's World

Hector's World is designed to help you learn about some important Artificial Intelligence techniques, while having a bit of fun developing a fantasy world and adventure stories.

You are given the basic Hector's World system. Your task is to enrich this skeleton, first as specified in the early exercises, and then in your own way. There is plenty of scope (and marks) for creativity.

### 1.1 The story

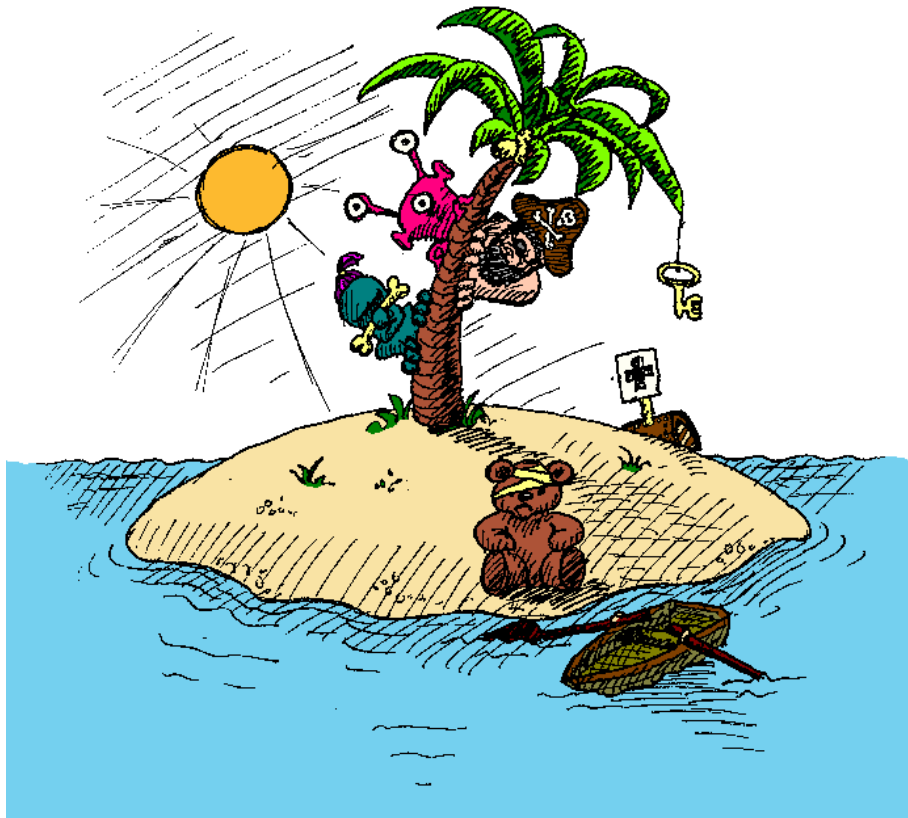


Figure 1: Picture of the tropical island

The story takes place on an enchanted island. Our hero, Hector the teddy bear, has been washed up at the Sea Cove. So that he doesn't die from a rather nasty tropical disease he picked up on his travels, he must locate the medicine chest, which is hidden on the island; he must then return to the Sea Cove. Fortunately Hector has a map, and knows where the chest is located. Unfortunately Lord Speke and

his henchmen don't take kindly to visitors to their island. They wander the island and will challenge Hector if they find him; when this happens, poor Hector has no choice but to flee in panic... unless he is carrying the shield, which is also hidden on the island (of course, he has to find this first).

However, all is not lost. Hector's beloved wife Hectorina died of the same tropical disease that plagues Hector, but she now roams the island as a ghost and tries to aid her love in any way she can. If she finds Hector, she will help him, either by flying him to the shield, or to the medicine chest.

Hector seeks the medicine chest, and returns to the Sea Cove when he has collected it; he does not seek the shield, but will pick it up if he finds it. He will move towards his goal location; however, when challenged he will flee in a random direction.

Hectorina, Lord Speke and the henchman wander randomly.

## 1.2 The software

The lab is written in JESS, the *Java Expert System Shell*. JESS is written in Java, and requires Java v1.3 (or higher). It is commercial software, though the Department has a free license for use in this lab. You may take a copy of JESS to work on the lab at home, but please do not use it for anything else, or we could get trouble with US government lawyers.

The JESS users manual, FAQ, mailing list archive and links to 3rd party JESS tools are all available from

<http://herzberg.ca.sandia.gov/jess/>  
which is linked from the CS1412 web page.

## 1.3 What you need

You will need for this lab

- This document, containing the exercises and additional information that you will need for them.
- Your own copy of the lab files - see the description of Exercise 1 in this manual.

By the end of this series of labs you should understand, and be able to use:

- Semantic networks and frames
- Production rules, "modules", conflict resolution strategies
- Simple text generation

You should have enough understanding of the software to be able to create a moderately complicated forward chaining system from scratch, though there won't be time to prove that in this lab unless you are unusually quick.

## 1.4 Marking

All exercises are marked, thus:

- Exercise 1 (1 scheduled session, should take less) 15 marks
- Exercise 2 (1 scheduled session, possibly less) 20 marks
- Exercise 3 (1 scheduled session) 25 marks
- Exercise 4 (2 scheduled sessions) 40 marks

In general, 75% of the marks for each exercise will be given for a minimal complete and correct answer - we call this the “baseline” mark. The remaining 25% are reserved to reward flair and creativity (which there is plenty of scope for, especially in Exercise 4.)

**WARNING** - The first lab is easy. The second ramps up a bit, the third and fourth are a lot harder. Be prepared for this.

**NOTE:** This typeface is used for commands which you should type into the system. The round brackets are part of the command. For example, you should type all five characters of

`(run)`

It is also used for code, and for system output.

This typeface is used for important concepts, and the names of parts of the JESS system, the first time they are mentioned.

Throughout this manual, “the program” means the files which make up the Hector’s World lab.

The lab exercises grow incrementally: each one builds on the one before. That is why there are *no* deadline extensions for the first two exercises, and why we strongly encourage you to keep backups at every stage.

**BEFORE you come into the lab for the start of each session, READ the complete description of the exercise. If you don’t, you will waste a lot of your time (and ours).**

Use `labmail` for all exercises.

**Plagiarism** is checked both automatically and manually. Last year several students were caught copying files from their friends and were heavily penalised.



Hector

## 2 The Exercises

### 2.1 Exercise 1: Welcome to Hectors World

(Lab session 1, week 2)

This exercise has three goals:

- To get Hector's World up and running;
- To understand the basics about the Hector's World system;
- To start extending the Knowledge Base.

Files to modify: `instances.clp`

#### 2.1.1 Installing and running the system

##### Installation

Before you can start the Hector's World exercises, you need to make your own copy of the files which make up the lab. To do this, you need to change to your home directory

```
cd ~
```

and then run the setup script

```
$CS1412/CS1412install
```

which is an alias for

```
/opt/info/courses/CS1412/CS1412install
```

This will make a directory called `CS1412` in your home directory, with subdirectories called `ex1`, `ex2`, `ex3` and `ex4` - you will need these to labmail your results. It will also copy the required files from `/opt/info/courses/CS1412` into each subdirectory. You will see a message telling you what is happening. You must do this on a PC running Linux, or it may not work.

##### Running the Program

To start up JESS for the first exercise, change to your `CS1412/ex1` directory and then type

```
java -cp /opt/info/courses/CS1412/Jess61_141 jess.Console
```

or

```
java -cp $CS1412/Jess61_141 jess.Console
```

This will start up JESS, inside a console called the **Listener**. This is where you load files, run the system, read error messages, see the results of a run, etc. The small prompt window at the bottom is where you type commands; the larger top window is where you see the output.

In the prompt at the bottom, type

```
(batch lab)
```

(including the brackets!) and press return. If you have copied the files and started up JESS in the correct directory, you should get the response 'TRUE' in the output, indicating that your command has returned successfully, i.e. the files have been loaded and put into the current memory.

To run Hector's World type

```
(run)
```

This will show you Hector's World, in action.

You are now ready to start.

(Note: If you try to "batch" a file which does not exist - for example if you mis-type `lab` - the Listener will freeze. If this happens, quit it (using the `close` button in the top right corner) and start again (`ctrl-P` in your working Linux window will save you typing the whole incantation again)).

When you are done, type (`exit`) to close the console.

### 2.1.2 Understanding the system

After Hector has escaped from the island and the run has ended, you can type (`reset`) to reset the knowledge base, and then you can re-run the program. Repeat this process a few times until you have an idea of what is happening. As doing this makes the system get gradually slower, you may have to exit and reload eventually.

What you see is the Hector's World `narrator` printing out what the `story creator` has come up with, this time round (random variables ensure that each time you play, the story is likely to be different).

You can step through the story with the command (`run n`), where `n` is the number of rules to fire in that step. There is a lot going on under the surface (especially in the initial set-up), but numbers bigger than 20 should give you visible results. There are typically 500 to 1000 rules fired in a completed lab.

Try the command (`facts`). It should give you a list of all the things in memory and their current state. You can use this to help understand and debug the program.

Now take a look at *still-map*, by typing (`batch map`), a tool which gives you a graphical representation of the island schematic. Note that this map is dynamic: if you change something, then the map will change accordingly (you will need to re-run (`batch lab`) and then the map to see any changes you make).

If you wish to watch the islanders and Hector during the run, type (`animap`) before (`run`) to see the *animated-map*. It may get distracting after a while, so use it if you need help understanding what is going on, but you won't need it all of the time! It can be closed with its own `close` button, its File menu, or the command (`noAnimap`). Unless you open the animated-map at the start of the Jess session, you may need to exit and reload to see it.

You should now have a basic understanding of how things work, at least on the surface. Section 3 of this document, the Reference Manual, explains things in more detail, but most of it is not necessary for this exercise.

### 2.1.3 Modifying the knowledge base

For this exercise you will be editing the `instances.clp` file.

You can use any editor you like to create and modify JESS files. If you use MSWord (*not* recommended), be sure to save the file as plain text, or it will contain control characters which will confuse JESS horribly.

Open up the `instances.clp` file, found in your `ex1` directory, and take a look.

#### Finish the map

Use the still-map, together with the Facts Viewer and the `instances.clp` file, to look at the map and understand how it is represented in the lab code.

Extend the map to the one below by adding new `LOCATION` facts. Look at the code for the existing Location facts and use it as a template for the new locations. (Section 4.1 (Facts) of the Reference Manual may be helpful.) You may also, if you like, change the names of the Locations, or, indeed, anything else. You will notice that the still-map has changed (once you have saved your changes and reloaded the map) to include the new Locations. You can use the still-map to check that you have made the changes correctly.

The map must be of "city block" form – no diagonal paths – the navigation mechanism can't deal with these. Also, take care with the `path` slot – it is a Boolean to tell the animated-map if a Location is a path (connects to exactly two other Locations) or a junction (connects to 2, 3 or 4 paths). Boolean values in Jess are in upper-case.

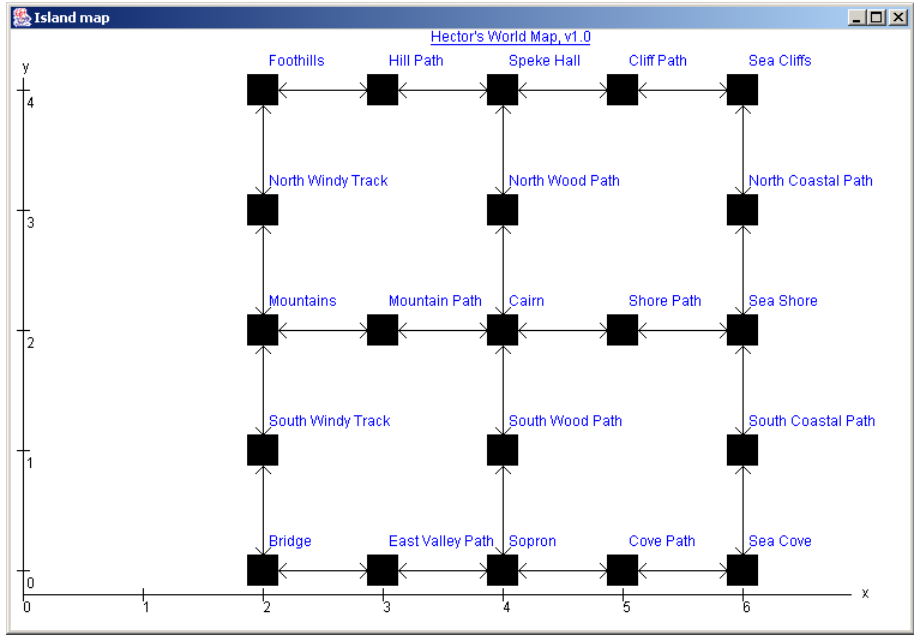


Figure 2: Schematic of the island map initially

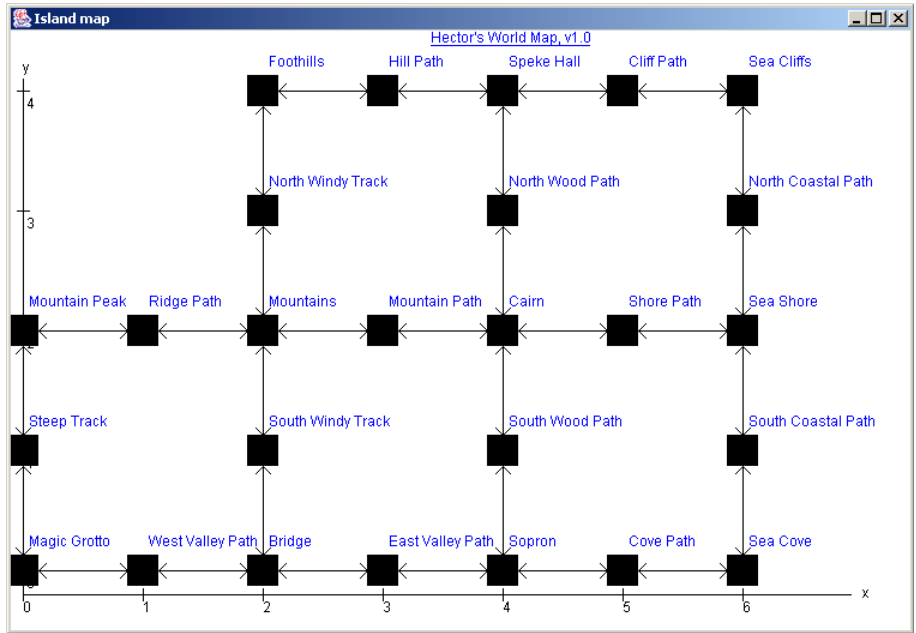


Figure 3: Schematic of the island map with new locations

## Create a new magic Thing

We want to add a new Thing, the magic **Key**, which will be important later. How do we do this? For the purposes of the story it will be sensible to make the key a fact of type **ESSENTIAL**. Look at the definition of **Medicine Chest**, and create the new fact **Key**. (Note: for the correct icons to be drawn on the animated-map, the text of the Key must end in "Key" and the Chest must end in "Chest".)

## Create a new Local agent

We want to create a new person, a villain **Hamish**. Again follow the template and create the new villain.

Make sure all your changes are fully commented (commented lines begin with a ; (semi-colon)).

Save a test run of your system (by selecting the output, copying it, and then pasting it into a text file), called **test1**, which shows your additions working. There is no need to use **labprint** in any of these exercises, so save that paper for a change!

### 2.1.4 Marking

- a Show the demonstrator your new Key and Villain, using the Facts viewer – 4 marks, including 1 mark for comments
- b Show the demonstrator your new locations, using the still-map viewer – 6 marks, including 1 mark for comments
- c Show the demonstrator your solution working, and explain what's happening – 5 marks
- d Labmail your code changes and **test1** (no need to **labprint**)

Total 15 marks (baseline 12).

It is not worth taking any time to get marks above the baseline on this exercise. If you finish it early, use the rest of the lab time to go ahead and start the next one.

**No extensions** will be given for this exercise, or the next one.



Magic Key

## 2.2 Exercise 2: To err is human...

(Lab session 2, week 4)

This exercise has three goals:

- To learn how rules provide behaviour in Jess;
- To learn how to extend the range of behaviour;
- To learn some useful debugging techniques.

Files to modify: `buggy1.clp`, `buggy2.clp`, `hero-spirit.clp`

You should do this exercise in your `ex2` subdirectory. This contains a clean copy of the original files, so if you want to keep your additions from the first exercise, you will need to copy your new, working `instances.clp` file from `ex1` to `ex2` (strongly recommended).

### 2.2.1 About rules and debugging

In a **rule based system**, rules fire when the facts which match their **preconditions** are present in the database. So, for example, the rule `meet_spirit` (in the file `hero-main.clp`) will fire when there is a teddy and a spirit at the same location. While this can be a really good way of doing things, letting us think in terms of facts and rules, it can make it difficult to track down bugs when they occur. For a start, the order in which the rules fire is not decided by the layout or structure of the code, but by the way the information in the system changes. Secondly, the actual circumstances in which a rule can fire may be rare, making it difficult to actually see the rule in operation. Jess, like most rule-based systems, partitions rules into **modules** to make the control flow more efficient and easier to understand. In this exercise, you will be developing the `HERO-SPIRIT` module (in the file `hero-spirit.clp`), which holds the rules for what happens when Hector meets Hectorina.

In fact most of the bugs in your code will be trivial typographic errors in spelling or punctuation, rather than logical or conceptual mistakes. (It is particularly easy to confuse `_` (underscore) with `-` (hyphen).) In the first part of this exercise, we have given you two rules with a few of these superficial bugs to fix - this will help you to learn about the structure of rules, and how to interpret Jess error messages. The second part will develop your conceptual understanding. You will need both these skills to get any further in the lab!

Before you start, read through section 3.2 of the Reference Manual, which explains the structure of rules in Jess.

### 2.2.2 Simple debugging

In your `CS1412/ex2` directory you should have the files `buggy1.clp` and `buggy2.clp`. Open an editor and load `buggy1.clp`, which contains the rule `help_hero_find_essential`. Can you see any mistakes?

Open a Jess console and load the lab, using the `(batch lab)` command. Run it once or twice to remind yourself how it works. Now `(reset)` and use the command `(batch buggy1.clp)` to load the `buggy1` rule. The listener window will give you an error message.

The error messages in Jess are not ideal - you will see a long list of errors leading down into the underlying Java system. The one line you need to look at is clearly left-indented. That line prints out the rule it was trying to load up, to the point at which it failed. Thus you know approximately which part of the rule contains the problem.

Use this to find out what is wrong and use the editor to fix it.

Save your corrected version of `buggy1.clp` and load it. Does it load? If not, you've either got a different problem or haven't fixed the original one. Try again.

(Remember: Even if your rules load correctly, there may still be errors. Don't believe that a rule works until you've actually seen it do what it's meant to.)

When you are happy with your rule, copy-and-paste it into the `HERO-SPIRIT` module defined in `hero-spirit.clp`.

Now do the same thing with the rule `to_the_sea_cove` in the file `buggy2.clp`.

### 2.2.3 Making a rule fire

So, now we have two rules that we think should work. The trouble is this: The rule only fires if Hector happens to bump into Hectorina. The reason for this is as follows: the rule `help_hero_find_essential` is in the module `HERO-SPIRIT` - it will only fire when we're in that module. Somewhere else in the program is a rule `meet_spirit`, which fires when Hector and Hectorina are in the same place, and changes the focus to the `HERO-SPIRIT` module. We could keep resetting and running, resetting and running, resetting and running... until they chance upon each other, but the lab is only two hours long! Instead we want to adjust the circumstances so that the rule fires more often.

There are lots of ways to do this, but these two are easiest:

- Make lots of Hectorinas so that Hector is much more likely to meet one. (Remember creating the villain Hamish?). This will of course work for shields, villains, keys, or whatever - we could just create lots of instances of `shield`, `shield1`, `shield2`, etc...
- Force the preconditions to occur by adding/changing another rule e.g., in this case, make Hectorina follow Hector around.

This latter approach is more elegant, so try to use this one.

The file `hero-action.clp` contains a rule `move`. The rule moves Hector on each turn. Find it, and try to get the gist of what's going on.

To make Hectorina follow Hector about, we want to move her to the same location as Hector each time we move him. Modify `move`, so that Hectorina goes wherever Hector goes. Changing this one rule alone will not suffice, as Hectorina will move of her own accord on the next turn *before* she has chance to meet Hector, (see the module control flow diagram), so you must look at `local.clp` and the rule `move_local`, and change it so that only Villains move, not all Locals (you only need to change one word here!).

(Note: If there are other instances of Spirit than Hectorina, they will appear as villains on the animated-map, but still behave as spirits.)

### 2.2.4 Simplifying the rules

Now that you have fully debugged the `help_hero_find_essential` and `to_the_sea_cove` rules, it's time for you to write a rule of your own which encompasses the two.

As you should have noticed from watching the objects as the system runs, Hector has a `to_go_to` attribute, or `slot`, which contains the name of the location which he needs to get to. Rather than having two rules, one to find essential objects and one for the Sea Cove, why not just have one rule which makes Hectorina fly Hector to his required destination.

Write a rule, `fly_teddy`, in the `HERO-SPIRIT` module, so that Hectorina will take Hector to his `to_go_to` location. Use the rules provided as a template to work from. (The action sent to the Narrator must be called `flies`, for reasons you'll learn in the next exercise.) Your new rule should be more general than the earlier rules, and therefore simpler. Note that many students always try to make this rule too complicated; it is shorter than either of the buggy rules.

(`fly_teddy` should be a separate rule in the `HERO-SPIRIT` module. It could be implemented as part of the `meet_spirit` rule when Hector meets Hectorina, but that then restricts any further development of

Hero/Spirit interaction in the system. When writing rules, always try and leave them as open as possible for future development. Also, modules are an important way of partitioning the search space: the system should consider as few rules as possible at any time, so you should keep all rules as restricted as possible.)

There will be two lines in `fly_teddy` to update the animated-map, (as in the buggy rules), worth one mark.

Save a test run of your system, called `test2`, which shows your additions working.

Finally (if you haven't already), make sure your code is fully commented. The next exercise starts again with a clean copy of the original files. You may well want to do something similar again, and it will be easier if you can look back and see what you did here.

### 2.2.5 Marking

- a Fix/Explain the errors in `buggy1.clp` – 4 marks
- b Fix/Explain the errors in `buggy2.clp` – 4 marks
- c Explain/Demonstrate your `fly_teddy` rule, using your `test2` file – 9 marks
- d Comments – 3 marks
- e Labmail your code changes and `test2`

Total 20 marks (baseline 15).

If you finish this exercise early, use the rest of the lab time to go ahead and start the next one.

No extensions will be given for this exercise. You cannot afford to get behind this early!



Hectorina

## 2.3 Exercise 3: Survival of the fittest

(Lab session 3, week 6)

The goal of this exercise is to develop substantially your ability to write rules in Jess.

Files to modify: `templates.clp`, `instances.clp`, `hero-villain.clp`, `lexitems.clp`

You will probably want your extended versions of `instances.clp` (from Ex. 1) and `hero-spirit.clp` (from Ex. 2) (as long as they're working properly).

Warning... this exercise is a lot harder than the last two. Be prepared for it.

### 2.3.1 More templates, facts and rules

In this exercise you begin to extend the program significantly. Currently when Hector meets a villain he will run away, unless he is holding a protector, in which case he can stand his ground. Now we want to give Hector the option of fighting against the Villains he meets if he's carrying a weapon. To expand this further we also want to have two types of Villains - Strong and Weak.

To do all this you will need to create three new classes – `WEAPON`, `STRONG-VILLAIN` and `WEAK-VILLAIN`, create at least one instance of a weapon, and change the current villains to be of either the type `strong` or `weak`. Finally, you will have to write new rules as part of the `HERO-VILLAIN` module.

The new villain classes should be subclasses which inherit from the currently existing `Villain` class.

The new `Weapon` class should be a subclass which inherits from the currently existing `THING` class. If the text of one of the weapons ends with "`Sword`", a sword icon will appear on the animated-map. All other weapons will just appear as text.

A suggested table of interaction between Hector and villains of different classes, based on whether he is carrying a protector or a weapon, is:

	<i>Weak Villain</i>	<i>Strong Villain</i>
No Protector, No Weapon	Hector flees	Hector flees
Protector, No Weapon	Hector stands	Hector flees
No Protector, Weapon	Hector fights	Hector stands
Protector, Weapon	Hector fights	Hector fights

You don't have to stick with this suggested set of interactions, but these are the ones for which the demonstrators will have model answers.

It is immediately obvious to write eight rules to match the above preconditions, more marks will be awarded if you can do it in fewer.

### 2.3.2 Keeping up with the action

You want to output some English text about Hector fighting in a similar way that there is text to Hector fleeing, moving etc. To do this, you need to add a new `LEXITEMS` rule for every new event type you add to the system, to convert the event from predicate form into "lexical items" (words), so that it can appear in the text.

If you look in the `text` file you will see a number of rules in the `LEXITEMS` module which take a predicate and create lexical objects. You should look at the current examples, and copy and modify them to add the new predicate form(s) you need to go with the actions in your new rules.

### 2.3.3 Marking

- Explain/Demonstrate the new template and instance definitions – 5 marks
- Explain/Demonstrate the new rules – 12 marks
- Explain/Demonstrate the new lexical rule(s) – 8 marks

d Labmail your code changes and `test3`

Total 25 marks (baseline 20).



Lord Speke

## 2.4 Exercise 4: Hector's Excellent Adventure?

(Lab sessions 4 and 5, weeks 8 and 10)

Now use everything you have learned, and your imagination, to extend Hector's World.

Again, your `ex4` subdirectory contains the original files, so copy forward any additions you are happy with and want to keep.

**Warning:** Be aware that the animated map may not keep up with all the additions from this point on.

### 2.4.1 Whatever next?

In this exercise we want you to add new activities to Hector's world. For most ideas, you need to create at least one new module, and you must create at least three new rules. Look at any of the existing modules (except MAIN, which is default) to see how to create a module. You must also create at least one new template and one new instance of that template. Bear in mind that, by now, we are looking for evidence that you can actually *use* both the underlying AI concepts and the JESS system: you will get higher marks for one really original character than for twenty identical villains named after Computer Science lecturers.

You have a choice to either develop two different ideas or just one but to a further level. So if you get stuck on the first design, you have the choice to try something else and still get credit for your existing work. You must try a different sort of problem for the second task, however, as shown in the diagram below.

Possible examples (which the demonstrators will have sample answers for):

- Thieves – A thief will steal weapons or protectors away from Hector, but not essentials. Further, Hector could also take essentials away from Lord Speke, (i.e. for example, the key to the medicine chest)
- Quests – available from specific or random locations, - if Hector is able to complete these then he is given artefacts. Further, these artefacts will help him, or even hinder him.
- Fighting by values – currently, Hector's reaction to meeting villains is limited and based on simple fact matching (holding weapon or not, weak or strong villain). Expand this by assigning strength to weapons and levels to villains
- Food and energy – Give Hector an energy level that decrements on each turn, and is replenished when he picks up food of varying goodness. If his energy drops low, he must look for food before the next goal. If he runs out of energy, he faints

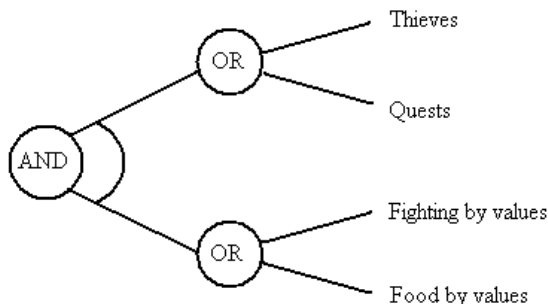


Figure 4: AND/OR Tree giving options available for Exercise 4

Parts a and b involve adding a new module and rules within it and other rules elsewhere. Parts c and d involve developing existing modules. So do not attempt two similar parts.

You do not have to use any of these suggestions, in fact variety and imagination are encouraged (they make the demonstrators' job more interesting too!). Bear in mind, however, that if you decide to go it on your own and create a complex system where all the characters build their own social circles and organise coffee mornings, the demonstrators will be impressed if it works, but will not necessarily find it easy to help you when something goes wrong!

### 2.4.2 Marking

If two parts have been completed, each is worth 30 marks (baseline 20). If one part has been completed to a further level, it is worth 40 marks (baseline 30). However, **the total marks available for this exercise is 40**, no matter how much work you have done, so don't try to go beyond the baseline on two parts.

Also `labmail` your code changes and a sample output in the file `test4`.

## 3 Error messages

### 3.1 “Compile-time” errors

The following errors may occur when calling (`batch lab`) to load the files into Jess.

*Message: No such variable teddy\_name.*

Using a variable that has not been declared correctly. Often the declaration is mistyped.

Example: Exercise 2: `help_hero_find_essential` in `buggy1.clp` has the variable `?teddy_nam` then `?teddy_name`.

*Message: text is not a multislot .*

If `text` is a slot in a fact, more than one atom has been entered before the closing bracket for that slot. Either there was a closing “” double quote missing, or a closing bracket missing. Example: Exercise 2: `to_the_see_cove` in `buggy2.clp` has the line `(text "Sea Cove)` with the missing closing double quote. (Jess reads that as one long string literal from the open double quote to the next open double quote.)

*Message: Expected ')’ .*

Exactly as it says, a closing bracket is missing.

Example: Exercise 2: `to_the_see_cove` in `buggy2.clp` has no closing bracket for the rule definition.

*Message: CE is a unary modifier not.*

There is a `not` function used, possibly on the LHS of a rule, and the closing bracket for it is missing.

### 3.2 “Run-time” errors

The following errors may occur while the lab is running and cause it to halt.

*Message: No such slot LOCATION in template MAIN::ESSENTIAL.*

1. Missing a closing bracket of a fact in the LHS of a rule in the file specified. In this case, the end of the `ESSENTIAL` fact in one of the rules has no closing bracket and sees the next fact `LOCATION` as a slot instead. This is from Exercise 2: `help_hero_find_essential` in `buggy1.clp`.
2. A slot name has been mistyped in either the file given or the `templates.clp` file.

*Message: No such variable essential\_name.*

Using a variable that has not been declared correctly.

Example: Exercise 2: `help_hero_find_essential` in `buggy1.clp`, this variable has an “\_” underscore where it is used but a hyphen “-” where it is declared. Must be consistent. Variable names (and rule names) in Jess are, by convention, in lower-case with underscores to separate the words. Function names are in lower-case with upper-case initials to separate the words.

*Rule trace: alternating NARRATOR::lexitems and LEXITEMS::continue*

There is a predicate applied which has not been matched by the `LEXITEMS` module. The rule `LEXITEMS::no_match` will prevent this infinite loop for narrations of the form `NOUN-VERB-NOUN`, but not `NOUN-VERB`.

## 4 Reference Manual

### 4.1 Facts

#### Overview

*Everything* in a JESS system is an **fact**. Facts which share important properties are grouped into **templates**. A fact is an **instance** of its template, and **inherits** any properties which are defined for the template as a whole. Actions are made to happen by rules giving instructions. You'll see some examples later on in this document.

#### Examples

The definition of a `location` fact has the general form:

1. `(LOCATION`
2.     `(name location_name)`
3.     `(text "Location text")`
4.     `(xGridRef int)`
5.     `(yGridRef int)`
6.     `(path Boolean)`
7.     `(access_to location1 location2 ... locationX))`

Here is a specific example:

```
(LOCATION
  (name Hill_Path)
  (text "Hill Path")
  (xGridRef 3)
  (yGridRef 4)
  (path TRUE)
  (access_to Foothills Speke_Hall)))
```

1 Make an instance of template `LOCATION`

2 with the name `location_name` - in this example, `Hill_Path`

3 which has a textual representation (for the text generator) - in this example, `"Hill Path"`,

4 and 5 with the X and Y grid coordinates of that Location - in this example, 3 and 4 - see the map - and

6 is for the animated-map, if this is a path (between exactly two locations) then `TRUE` or a junction (connects paths) then `FALSE`

7 has access to `location1` and `location2`, i.e. you can get there from here - in this example, the Foothills and Speke Hall. There can be as many or as few locations in the access list as you wish.

Here is an example definition of a (small) group of facts :

1. `(deffacts PEOPLE`
2.     `(TEDDY`
3.         `(name Hector)`
4.         `(text "Hector")`
5.         `(proper TRUE))`
6.     `(SPIRIT`
7.         `(name Hectorina)`
8.         `(text "Hectorina")`
9.         `(proper TRUE)))`

1 Introduces a group of definitions of facts, in this case called **PEOPLE**.

2 Make an instance of the template **TEDDY**

3 Which can be referred to by the name **Hector**,

4 named in the text output as “Hector” (note that the name used in the text output does not have to be the same as the *facts name* used by the system. Your teddy could perfectly well have the instance name “Bobby” and the text name “the big fat pink bear with yellow spots”. You can take advantage of this to make your stories more interesting to read, but don’t go overboard.)

5 The **proper** slot is used by the text generator. A fact which is **proper FALSE** needs a “determiner” with its name, one which is **proper TRUE** does not. Thus we want the story to talk about “the key” and “the shield” but not “the Hector”.

Note: The **active** slot (not defined above) is crucial to the overall control of the world, which will be described below. All **PERSONs** start out as not-active and will be activated as soon as the game cycle starts.

6-9 are parallel to 2-5, but notice the extra closing bracket at the end of line 9, which matches the opening bracket of the group definition in line 1.

## 4.2 Rules

### Overview

Jess is a modern re-implementation of the classic "expert system shell", or "production rule system", developed in the early days of Artificial Intelligence. Rules are made up of conditions ("left hand sides", LHS) and actions ("right hand sides", RHS). The conditions state a set of facts about the world (as known to the program) which must be true for the rule to fire. The actions specify an action or actions to be carried out if the preconditions are met. (See the notes for lecture 3 for further detail on Production Rule Systems.)

Anything named in the actions of a rule must first be named in the preconditions. Whenever you write a new rule, make a systematic check to be sure that everything involved in the actions has been declared in the preconditions. (Check also that everything declared in the preconditions is actually used in the actions:if not, it doesn't need to be there.)

Conflict resolution in JESS is handled by **salience** (other systems sometimes call this "priority"). Look at the existing rules and to see how you use salience to give a rule a higher (or lower) priority than normal. For example, the **HERO-MEETING** module contains two rules, one which will fire when Hector meets a spirit and one for when he meets a villain. But what happens if he meets a villain and a spirit at the same time?

```
(defrule meet_spirit
  (declare (salience 1))
  (TEDDY
  ...
```

```
(defrule meet_villain
  (TEDDY
  ...
```

A rule without a declared salience is assumed to have a salience value of 0. So the `meet_spirit` rule will fire first, and Hectorina will fly Hector away before the villain can challenge him.

### Examples

Here is the `move` rule, in `hero-action.clp`, which you may want to modify in Exercise 2:

```
1. (defrule move
2.   ?teddy <- (TEDDY
3.     (name ?teddy_name)
4.     (text ?teddy_text)
5.     (active TRUE)
6.     (at ?current_location)
7.     (to_go_to ?goal_location))
8.   (LOCATION
9.     (name ?goal_location)
10.    (x_grid_ref ?goal_x_ref)
11.    (y_grid_ref ?goal_y_ref))
12.  (not (NARRATION (predicate moves)))
13.  =>
14.  (bind ?location_to_move_to (getLocationInDirectionOfGoal
                               ?current_location ?goal_x_ref ?goal_y_ref))
15.  (tellNarrator moves ?teddy_name ?location_to_move_to)
16.  (mapMoveAllPeople ?teddy_text ?location_to_move_to)
17.  (mapSetTeddyNormal)
18.  (modify ?teddy (at ?location_to_move_to) (active FALSE)))
```

1 Introduces a rule, in this case called `move`

There is no declaration of salience, so it defaults to 0

2 which involves an entity bound to the variable `?teddy`, an instance of the class, or template, `TEDDY`

3 with a name bound to the variable `?teddy_name`

4 with the text string bound to the variable `?teddy_text`

5 with the value of active slot set to `TRUE` - in other words he is ready to move - see the description of the game cycle in Section 4.1 for a full explanation of this

6 at a location bound to the variable `?current_location`

7 with the value of `to_go_to` (the place he's trying to get to) bound to the variable `?goal_location`

8 The rule also involves an entity of class `LOCATION`

9 with a name bound to the variable `?goal_location`

10 with the value of `x_grid_ref` (the x-coordinate on the map grid) bound to the variable `?goal_x_ref`

11 and the value of `y_grid_ref` (the y-coordinate on the map grid) bound to the variable `?goal_y_ref`.

12 The rule also involves an entity which is the piece of narration asserted after the rule has been fired. The logical operator `not` means that the rule cannot fire again if the described instance is present. The line is to stop the rule from firing repeatedly when reached, although the `(not (NARRATION ...))` construct is more important in other modules, such as `HERO-VILLAIN`

13 If all these preconditions are met, then do this:

14 bind the value of the variable for the place Hector should move to, to the one chosen by the navigation mechanism (you can regard this as a bit of black magic if you wish)

15 add to the narrator the fact that the teddy moves to the chosen next place, at the current place in the story sequence

16 tell the animated-map that the people are moving, and the one who initiated the move has the text `?teddy_text` and is moving to `?location_to_move_to`

17 tell the animated-map that Hector is no longer flying or fighting or fleeing, he is in a normal state

18 change the teddy to make him be at the next place, and shut him down so the rest of the cycle can go ahead. Note you have to do this explicitly, it's not enough just to tell the narrator about it.

## 4.3 Modules

### Understanding JESS modules

In JESS, modules are used for two reasons:

- 1 To make control flow easier
- 2 To partition the search space

This section will focus on part 1. : using modules to control the flow of execution.

The first thing about modules is that rules **belong** to a module. If you create a new module, then any rules you create after that belong to the new module (until another new module is created).

The second thing about modules is that rules that belong to a module, will **only** fire if that module has been given execution control (called 'focus'). The main module starts off with focus, and is allowed to give it away to other modules.

The third and final thing about modules is that when that module is called (and unless another 'focus' command is given by one of the rules in that module) then all rules that belong to that module which have been activated **will** fire (i.e. execute), the conflict resolution strategy will determine which order the rules inside the **focus module** will fire. After they have fired, and once there are no more rules in the **focus module** to fire, then the focus will return to whatever module called it. In effect, modules are operated like a stack. When a module is focussed on, it is pushed to the top of the stack, when there are no more rules in that module to fire, it will be popped from the stack.

For example, the **HERO-MEETING** module (from the `hero-meeting.clp` file which we looked at above) has two rules which belong to it: `meet_spirit`, and `meet_villain`. The `meet_spirit` rule passes focus to the **HERO-SPIRIT** module. The meat filling rule passes focus to the **HERO-VILLAIN** module. So, when Hector meets a spirit, the `meet_spirit` rule fires, passing control to the **HERO-SPIRIT** module. The rules belonging to that module (which you added in Exercise 2) are now available to fire.

After the chosen rules in **HERO-SPIRIT** have fired, control returns to **HERO-MEETING**. If another rule there can be fired, it will be. Otherwise, control will return to the place it came from - the **HERO-MAIN** module - where Hector again has two possibilities, meeting another local, or going on towards his goal.

## 4.4 Hector's World Modules and Rules

### 4.4.1 The Story Creator

#### The WORLD Module

Saliency 1	<code>local_turn</code>	Play each LOCAL in the LOCAL module.
Saliency 0	<code>hero_turn</code>	Play the TEDDY in the HERO-MAIN module.

#### The LOCAL Module

Saliency 1	<code>move_local</code>	Move each LOCAL in turn.
Saliency 0	<code>check_local</code>	Check to see if a LOCAL has met the TEDDY

#### The HERO-MAIN Module

Saliency 1	<code>meet_local</code>	If a LOCAL has found the TEDDY, take care of these events in the HERO-MEETING module.
Saliency 0	<code>play_hero</code>	If TEDDY is still able to, he will play, in the HERO-ACTION module.

#### The HERO-MEETING Module

Saliency 1	<code>meet_spirit</code>	If a SPIRIT has found the TEDDY, she helps him, then focus passes to the HERO-SPIRIT module.
Saliency 0	<code>meet_villain</code>	If a VILLAIN has found the TEDDY, it challenges him, then focus passes to the HERO-VILLAIN module.

#### The HERO-ACTION Module

Saliency 4	<code>see_thing</code>	If there is a THING (PROTECTOR or ESSENTIAL), the TEDDY will see it.
Saliency 3	<code>take_thing</code>	If there is a THING, the TEDDY will pick it up, which will inactivate him and automatically return focus to HERO-MAIN.
Saliency 2	<code>set_goal</code>	If TEDDY has no goal (place to go to), give him one by passing focus to HERO-GOAL.
Saliency 1	<code>check_goal</code>	If TEDDY has achieved his goal (at the location to go to), remove that goal. The <code>set_goal</code> rule will automatically fire next.
Saliency 0	<code>move</code>	If none of the above rules have been fired, TEDDY will move one step towards his goal location, then focus will return to HERO-MAIN.

#### The HERO-SPIRIT Module (rules from buggy files)

Saliency 0	<code>help_hero_</code> <code>find_essential</code>	If there is an ESSENTIAL which is not carried, move to it.
Saliency 0	<code>to_the_sea_cove</code>	If all ESSENTIALs are carried, move to the Sea Cove.

#### The HERO-VILLAIN Module

Saliency 0	<code>flee</code>	Flee randomly if no PROTECTOR is carried.
Saliency 0	<code>stand</code>	Stand ground if a PROTECTOR is carried.

#### The HERO-GOAL Module

Saliency 0	<code>essential</code>	If there is an ESSENTIAL which has not been picked up and TEDDY has no goal, make the goal to get that ESSENTIAL.
Saliency 0	<code>escape</code>	If there are no ESSENTIALs which have not been picked up and TEDDY has no goal, make the goal to escape to an <code>entry_exit_point</code> .
Saliency 1	<code>finish</code>	If there are no ESSENTIALs which have not been picked up and TEDDY has reached the <code>entry_exit_point</code> , he can finish the adventure.

The MAIN, NARRATOR and HERO-GOAL modules should not need to be modified, except possibly for advanced attempts at Exercise 4.

#### 4.4.2 The Text Generator

##### The NARRATOR Module

Saliency 0	lexitems	For each LEXITEM in the system, increment the NARRATOR-COUNTER and pass focus to the LEXITEMS module.
Saliency 0	last_sentence	If there is a SENTENCE and the NARRATOR-COUNTER is complete, print and retract the SENTENCE and reset the NARRATOR-COUNTER.

##### The LEXITEMS Module

Saliency 0	continue	When all the current predicates have been turned into lexical items, pass focus to the LEX-RULES module.
Saliency 0	XXX	Create the lexical items for predicate XXX.
Saliency -5	no_match	If there is an unrecognised predicate, discard it.

##### The LEXRULES Module

Saliency 10	check_determiner	Add a determiner if required. Get the text for the LEXITEM
Saliency 10	print_previous_sentence	Leave unchanged.
Saliency 5	create_verb_phrase	Create a VERBPHRASE from a VERB and an object NOUN.
Saliency 5	create_sentence	Create a SENTENCE from an agent NOUN and a VERBPHRASE.
Saliency 0	create_simple_sentence	Create a SENTENCE from an agent NOUN and a VERB.
Saliency 0	make_normal_sentence_into_head_of_sentence	Convert a SENTENCE to a head SENTENCE if there is a LINK.
Saliency 0	make_full_sentence	Create a SENTENCE from a head SENTENCE, a LINK and a SENTENCE.

#### 4.4.3 The Top Level

##### The MAIN Module

Saliency 15	setup_animated_map	Call the animated map, even if it is not being used.
Saliency 10	collect_locations	Load the locations into a single data type to use later.
Saliency 5	setup_teddy	Place the TEDDY at the starting point.
Saliency 5	setup_locals	Place the LOCALS in random places.
Saliency 5	setup_things	Place the THINGS in random places.
Saliency 0	start_up	Begin the game, pass focus to the SYSTEM module.
Saliency 0	shut_down	End the game when there is nothing else to say.

##### The SYSTEM Module

Saliency 3	tell_story	First, the story from the previous turn is told, focus passes to NARRATOR module.
Saliency 2	check_teddy	If the TEDDY has not achieved all his goals, allow SYSTEM to continue
Saliency 1	reactivate	If there is a PERSON who has not moved yet (they are active FALSE), and the previous rule says TEDDY has not achieved all his goals, make each PERSON able to play (active TRUE). This rule is called once per PERSON.
Saliency 0	go_on	When all the PERSONS are able to move (higher saliency rule), pass focus to the WORLD module to create the story for the next turn. Also increments the TURN-COUNTER.

## 4.5 Map functions

There are two dynamic maps in Hector's World. The *still-map* shows just the `LOCATION`s and the access paths between them. It can be called between runs of the game. The *animated-map* shows the `PERSON`s and `THING`s as they move around the island, and displays the major actions that occur. It runs at the same time as the game.

### 4.5.1 Still-map

To show the still-map, type `(batch map)` at the Jess console command line. You cannot move it from its initial location without losing the drawing, as it does not repaint when moved. It can be closed with its close window icon, or just left open (this will not drain the system of any noticeable resources). To continue with the lab, type `(batch lab)` again.

It is formed by running the batch file called `map`, which uses `island.clp`, `templates.clp` and `instances.clp` to find out which `LOCATION`s have been declared, then draw them using Java GUI classes access entirely by Jess code.

### 4.5.2 Animated-map

To show the animated-map, type

```
(batch lab)
(animap)
(run)
```

at the Jess console command line. It can be relocated to any place on the desktop, and can be minimised then restored. The close button will destroy the map, but the lab will still run (after a few seconds delay). The command `(noAnimap)` will destroy the map window in the same way. There is a loading time of a few seconds, which is a good chance to locate the window as desired.

It is common for a Java Out Of Memory error to occur when using the animated- map, but it can be ignored. If many calls to `(batch lab)` are made, the animated- map will slow things down even if not used; in this case, it is quicker to exit Jess and start again. To exit, type `(exit)` or find it in the File menu of the map window.

The map window has an option to change the speed of the animations, an option to show a text box at the bottom with the sentences made by the story-teller, and a pretty background image instead of the blue and green blocks. (The image is of Middle Earth from *The Lord Of The Rings*, but mirrored to match the east coast of Hector's World (artist unknown).

The animated-map was written in Java by Aliya Aziz as part of her Third Year Project 2004, and used here with kind permission. The Java code is accessed by Jess via a middle layer, in the file `mapcreator.clp`.

## 4.6 Debugging functions

### `(facts)`

Entered from the Jess console command line, this will list all of the facts currently in the system, and their values. The facts can be modified from the command line; the `modify` command takes a `fact-id` as its first argument, which is something like the first token of `?teddy <- (TEDDY ...)` in a rule LHS, or just the integer from the `fact-id` (given in the `(facts)` output) when using the command line.

### `(halt)`

Inserted in a function or rule RHS, this will halt any run when reached. At that point, a debugging command could be entered on the command line to investigate. The `halt` command would usually be inserted just before a faulty piece of code. Note that it does not halt the system immediately, but at the end of the rule that is currently being fired. Another call to `(run)` after will continue. Remember that `(run)` can take an argument of how many rules to fire.

### `(list-focus-stack)`

Entered at the command line, this will print out the stack of all the modules to show which module is currently in focus on top. Or more simply, `(get-focus)` will print just the current module in focus.

### `(printout t <string-expression> crlf)`

Entered on command line or in functions or rule RHS, this will print whatever it is given to the Jess console. The token `crlf` stands for a carriage return specific to the operating system in use. Any number of them can be used, although there is none at the end by default. The string expression can be any variables or literals, where literals are given in "double quotes" and they are all separated by white space and **not** plus symbols as in Java. For example, `(printout t "The value of ?var is " ?var " at this time." crlf)`

### `(watch <watchable>)`

### `(unwatch <watchable>)`

Entered on the Jess console command line, this will print out additional debugging information. Switch on with `watch` and off with `unwatch`. `<watchable>` must be one of the following (in decreasing order of usefulness):

- `rules`: prints out the name of each rule fired
- `facts`: prints a message when a fact is asserted, modified or retracted
- `focus`: prints a message when there is a change of module focus
- `activations`: prints a message when a rule is activated (ready to fire)
- `compilations`: prints a message when a rule is compiled
- `all`: all of the above

Any number of these can be watched by multiple calls to `watch`.

## 4.7 Jess console keyboard shortcuts

A modified Jess console exists in `$CS1412/Jess61_141`, improved by Richard Craggs over the regular Jess console, in `$CS1412/Jess61`. It was modified specifically for this lab and should be used in preference.

The modifications are the introduction of a number of keyboard shortcuts and their functions, listed below.

- Cursor keys  $\uparrow$  up and  $\downarrow$  down: scroll through history of command line entries to the console
- F1: watch/unwatch rules
- F2: watch/unwatch focus stack
- F3: watch/unwatch agenda when in HERO-\* modules
- F4: watch/unwatch agenda when in LEXITEMS and LEXRULES modules
- F5: watch/unwatch agenda at all times

The output from the watching by the F-keys will produce text on the terminal window used to run the Java Jess console, and not on the Jess console screen as with typed entries to the `watch` functions on the command line.

## 5 The Hector Files

### 5.1 Overview of file structure

#### 5.1.1 Story Creator and Text Generator

The program is broken down into two main elements, the **Story Creator** and the **Text Generator**.

##### The Story Creator

This is the abstraction for Hector's World itself. The story creator plays the game, it controls all the agents, it makes all the moves, it determines what happens, and when. The story creator is not externally visible to the user when the program is executed.

##### The Text Generator

This is the link between human user and computer bits. The **Text Generator** tells the story - it tells whatever the **Story Creator** does, and it does so by converting from the internal representation of the story into natural language which can be viewed on screen.

The **Story Creator** helps in this process by earmarking the interesting snippets of information to tell, so that all the **Text Generator** has to do is to collect these snippets and translate them one at a time.

##### The link between the two

There are, in fact, two links. The first is a shared knowledge base, which the **Story Creator** writes to and the **Text Generator** reads from.

The second is the game **System**, which plays out Hector's World like a turn-based game. It lets the **Text Generator** start the story off, then it lets the **Story Creator** play one turn, then it lets the **Text Generator** tell what happened in that turn, and so on until the story is over (at which point the System shuts down).

#### 5.1.2 Architecture: Modular structure

The system architecture is flexible, and it can be modified. Each module is self contained in that all data invariants for that module are defined. This means that a module can be expanded, split into two, or two modules can be condensed into one without considering anything else.

##### Modules: functional decomposition

The modules have been broken down by their functional abstraction (i.e. purpose), listed below:

- **main**: initialises the system based on random variables, and shuts down the system once Hector has escaped. Also contains global functions like `getRandom`, which can then be called from any module. After initialisation, it delegates control to the **system** module.
- **system**: is the controller module. It switches control back and forth between the **Story Creator** and the **Text Generator**, until Hector escapes, when it terminates and returns to **main**. Its control lies in its power to activate (or re-activate) the agents.
- **world**: is the incarnation of the story creator. It represents Hector's World. When called, It plays each agent (Hero, Villain and Spirit) once, then terminates and returns to **system**. It first plays the Locals, then plays the Hero - this is done by delegating control to the appropriate modules.
- **local**: is responsible for playing all active locals once each, and inactivating them once they have played. Upon completion of this, control will return to **world**.

- **hero-main**: is responsible for playing the Hero. The behaviour of the Hero is complex, and so it has been split up according to functionality. If Hector finds himself face to face with one of the locals (Villain or Spirit) then control will be delegated to **hero-meeting**, which will determine what to do. Otherwise, it will delegate control to **hero-action**, which will determine what action to take. Once the Hero has moved, **hero-main** has nothing left to do, and so it terminates and returns to **world**.
- **hero-meeting** will determine what to do when Hector comes face to face with one of the locals. If the local is a spirit then it will delegate control to **hero-spirit** to determine how to deal with the spirit (Hectorina), otherwise (the local must be a villain) control is delegated to **hero-villain** to determine how best to deal with the villain. In the case where there is both spirit and villain and Hero, the spirit will have priority. Control will return to **hero-main** once either the Hero is no longer active or else there are no more events (meetings with locals) to process.
- **hero-action** will make a move for Hector, assuming that he has not been forced to make one already this turn by one of the locals. Once that action is made, control will return to **hero-main**. If there is an interesting object for Hector to pick up, then he will take it, otherwise he will move according to his goal destination. If Hector does not know where to go next, then **hero-goal** will be invoked to tell him where he wants to go next. Note that escaping counts as a move, it is not something which happens automatically. Once Hector has moved control will return to **hero-main**.
- **hero-goal** will determine where the Hero should be headed next, based upon the current context.
- **narrator** is the text generator. This module has a dual purpose, firstly it controls the execution of the story telling by making sure that only one item at a time is told, and secondly when there is nothing more to tell it will finish off the story telling and return to **system**. When there is something to say, it will call **lexitems** which will know what to do with it.
- **lexitems** extricates the meaning of whatever the **narrator** wants to say next, and assembles it into a primitive lexical form which can then be assembled into phrases and sentences. This is done by **lexrules**, once **lexitems** has figured out what to say, which it will delegate control to.
- **lexrules** is the grammar tool. It puts together the various bits that **lexitems** wants it to say into natural language. Once **lexrules** has finished doing this, control will return to **lexitems**, which itself will have nothing more to do, and so control will return back to **narrator** which will decide whether there is something more to say or not.

### 5.1.3 Modules: control flow

The control flow between modules is shown in the diagram below:

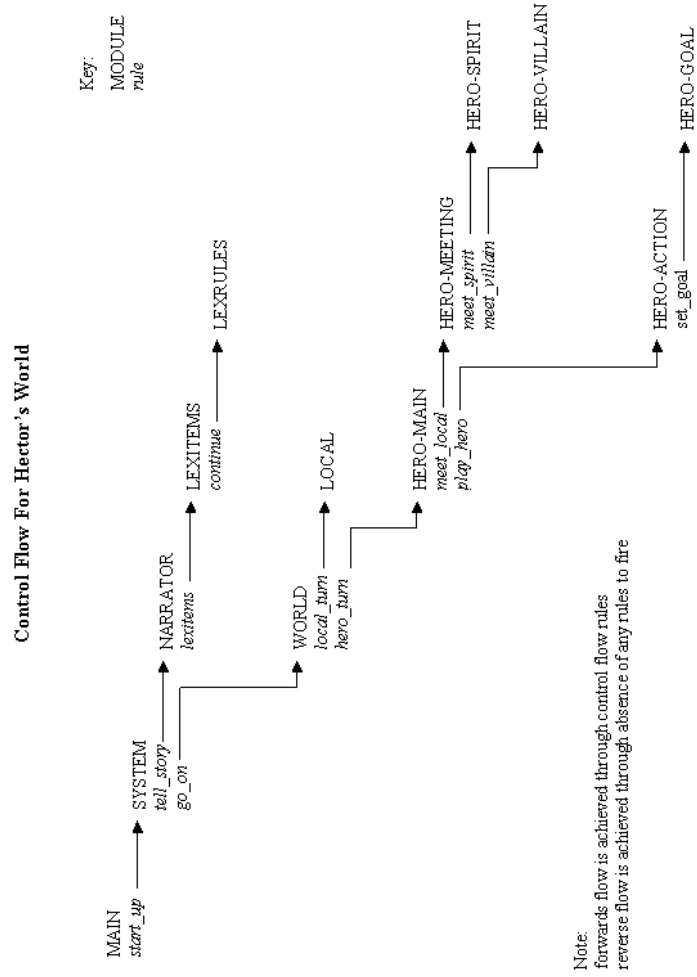


Figure 5: Module flow diagram

### 5.1.4 Template structure

The structure of the templates used is given below, as UML Class Diagrams:

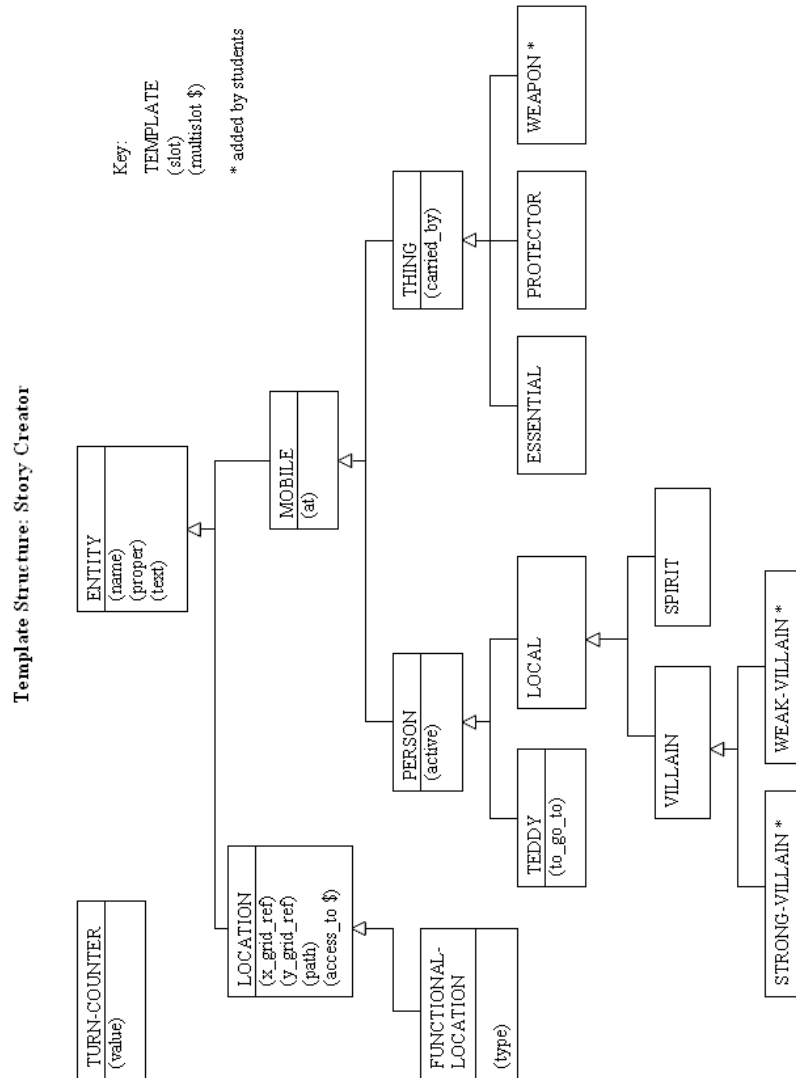


Figure 6: Story Creator templates

Template Structure: Text Generator

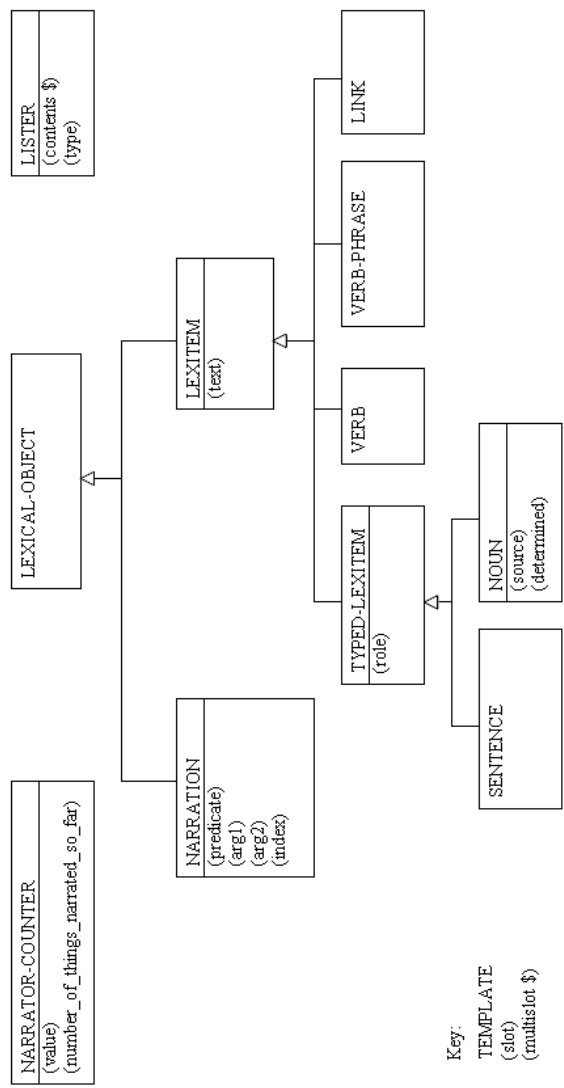


Figure 7: Text Generator templates

## 5.2 Code listings

### 5.2.1 lab

```
(clear)

(batch templates.clp) ; classes
(batch instances.clp) ; things

(batch main.clp)      ; sets up and shuts down the world

(batch hero-goal.clp) ; overall driver for the teddy

(batch system.clp)   ; controls the story creator / teller cycle
(batch world.clp)    ; controls the local / hero cycle
(batch local.clp)    ; moves the locals

(batch hero-main.clp) ; Has the teddy met a local?
(batch hero-meeting.clp) ; If so, spirit or villain?
(batch hero-spirit.clp) ; rules after meeting a spirit
(batch hero-villain.clp) ; rules after meeting a villain
(batch hero-action.clp) ; rules if no local met

(batch narrator.clp) ; tells the story
(batch lexitems.clp) ; words
(batch lexrules.clp) ; grammar rules

(set-current-module MAIN)

(reset)
```

## 5.2.2 templates.clp

```
(defacts instances

; All facts defined here instantiate templates from the
; 'templates.clp' file.

;;;;;;;;;;;;;;;;;;
;; LOCATIONS ;;
;;;;;;;;;;;;;;;;;;

(LOCATION
  (name Foothills)
  (text "Foothills")
  (x_grid_ref 2)
  (y_grid_ref 4)
  (path FALSE) ;; Foothills is not a path it is a regular location
  (access_to Hill_Path N_Windy_Track))

(LOCATION
  (name Hill_Path)
  (text "Hill Path")
  (x_grid_ref 3)
  (y_grid_ref 4)
  (path TRUE)
  (access_to Foothills Speke_Hall))

(LOCATION
  (name Speke_Hall)
  (text "Speke Hall")
  (proper TRUE)
  (x_grid_ref 4)
  (y_grid_ref 4)
  (path FALSE)
  (access_to Hill_Path N_Wood_Path Cliff_Path))

(LOCATION
  (name Cliff_Path)
  (text "Cliff Path")
  (x_grid_ref 5)
  (y_grid_ref 4)
  (path TRUE)
  (access_to Speke_Hall Sea_Cliffs))

(LOCATION
  (name Sea_Cliffs)
  (text "Sea Cliffs")
  (x_grid_ref 6)
  (y_grid_ref 4)
  (path FALSE)
  (access_to Cliff_Path N_Coastal_Path))

(LOCATION
  (name N_Windy_Track)
  (text "North Windy Track")
  (x_grid_ref 2)
  (y_grid_ref 3))
```

```
(path TRUE)
(access_to Foothills Mountains))
```

```
(LOCATION
(name N_Wood_Path)
(text "North Wood Path")
(x_grid_ref 4)
(y_grid_ref 3)
(path TRUE)
(access_to Speke_Hall Cairn))
```

```
(LOCATION
(name N_Coastal_Path)
(text "North Coastal Path")
(x_grid_ref 6)
(path TRUE)
(y_grid_ref 3)
(access_to Sea_Cliffs Sea_Shore))
```

```
(LOCATION
(name Mountains)
(text "Mountains")
(x_grid_ref 2)
(y_grid_ref 2)
(path FALSE)
(access_to N_Windy_Track Mountain_Path S_Windy_Track))
```

```
(LOCATION
(name Mountain_Path)
(text "Mountain Path")
(x_grid_ref 3)
(y_grid_ref 2)
(path TRUE)
(access_to Mountains Cairn))
```

```
(LOCATION
(name Cairn)
(text "Cairn")
(x_grid_ref 4)
(y_grid_ref 2)
(path FALSE)
(access_to Mountain_Path N_Wood_Path Shore_Path S_Wood_Path))
```

```
(LOCATION
(name Shore_Path)
(text "Shore Path")
(x_grid_ref 5)
(y_grid_ref 2)
(path TRUE)
(access_to Sea_Shore Cairn))
```

```
(LOCATION
(name Sea_Shore)
(text "Sea Shore")
(x_grid_ref 6)
(y_grid_ref 2)
(path FALSE)
```

```

(access_to Shore_Path S_Coastal_Path N_Coastal_Path))

(LOCATION
  (name S_Windy_Track)
  (text "South Windy Track")
  (x_grid_ref 2)
  (y_grid_ref 1)
  (path TRUE)
  (access_to Mountains Bridge))

(LOCATION
  (name S_Wood_Path)
  (text "South Wood Path")
  (x_grid_ref 4)
  (y_grid_ref 1)
  (path TRUE)
  (access_to Cairn Sopron))

(LOCATION
  (name S_Coastal_Path)
  (text "South Coastal Path")
  (x_grid_ref 6)
  (y_grid_ref 1)
  (path TRUE)
  (access_to Sea_Shore Sea_Cove))

(LOCATION
  (name Bridge)
  (text "Bridge")
  (x_grid_ref 2)
  (y_grid_ref 0)
  (path FALSE)
  (access_to S_Windy_Track E_Valley_Path))

(LOCATION
  (name E_Valley_Path)
  (text "East Valley Path")
  (x_grid_ref 3)
  (y_grid_ref 0)
  (path TRUE)
  (access_to Bridge Sopron))

(LOCATION
  (name Sopron)
  (text "Sopron")
  (proper TRUE)
  (x_grid_ref 4)
  (y_grid_ref 0)
  (path FALSE)
  (access_to E_Valley_Path S_Wood_Path Cove_Path))

(LOCATION
  (name Cove_Path)
  (text "Cove Path")
  (x_grid_ref 5)
  (y_grid_ref 0)
  (path TRUE)

```

```

(access_to Sopron Sea_Cove))

(FUNCTIONAL-LOCATION
(name Sea_Cove)
(text "Sea Cove")
(type entry_exit_point)
(x_grid_ref 6)
(y_grid_ref 0)
(path FALSE)
(access_to Cove_Path S_Coastal_Path))

;;;;;;;;;;;;;
;; PEOPLE ;;
;;;;;;;;;;;;;

(TEDDY
(name Hector)
(text "Hector")
(proper TRUE))

(VILLAIN
(name Lord_Speke)
(text "Lord Speke")
(proper TRUE))

(VILLAIN
(name Hughie)
(text "Hughie")
(proper TRUE))

(SPIRIT
(name Hectorina)
(text "Hectorina")
(proper TRUE))

;;;;;;;;;;;;;
;; THINGS;;
;;;;;;;;;;;;;

(ESSENTIAL
(name Chest)
(text "Medicine Chest"))

(PROTECTOR
(name Shield)
(text "Shield"))

;;;;;;;;;;;;;
;; ENGINE INSTANCES ;;
;;;;;;;;;;;;;

(LISTER (type locations))

(NARRATOR-COUNTER (value 0)(number_of_things_narrated_so_far 0))

```

```
(TURN-COUNTER (value 0))
```

```
)
```

### 5.2.3 instances.clp

```
(deffacts instances

; All facts defined here instantiate templates from the
; 'templates.clp' file.

;;;;;;;;;;;;;;;;;;
;; LOCATIONS ;;
;;;;;;;;;;;;;;;;;;

(LOCATION
  (name Foothills)
  (text "Foothills")
  (x_grid_ref 2)
  (y_grid_ref 4)
  (path FALSE) ;; Foothills is not a path it is a regular location
  (access_to Hill_Path N_Windy_Track))

(LOCATION
  (name Hill_Path)
  (text "Hill Path")
  (x_grid_ref 3)
  (y_grid_ref 4)
  (path TRUE)
  (access_to Foothills Speke_Hall))

(LOCATION
  (name Speke_Hall)
  (text "Speke Hall")
  (proper TRUE)
  (x_grid_ref 4)
  (y_grid_ref 4)
  (path FALSE)
  (access_to Hill_Path N_Wood_Path Cliff_Path))

(LOCATION
  (name Cliff_Path)
  (text "Cliff Path")
  (x_grid_ref 5)
  (y_grid_ref 4)
  (path TRUE)
  (access_to Speke_Hall Sea_Cliffs))

(LOCATION
  (name Sea_Cliffs)
  (text "Sea Cliffs")
  (x_grid_ref 6)
  (y_grid_ref 4)
  (path FALSE)
  (access_to Cliff_Path N_Coastal_Path))

(LOCATION
  (name N_Windy_Track)
  (text "North Windy Track")
  (x_grid_ref 2)
  (y_grid_ref 3))
```

```

(path TRUE)
(access_to Foothills Mountains))

(LOCATION
(name N_Wood_Path)
(text "North Wood Path")
(x_grid_ref 4)
(y_grid_ref 3)
(path TRUE)
(access_to Speke_Hall Cairn))

(LOCATION
(name N_Coastal_Path)
(text "North Coastal Path")
(x_grid_ref 6)
(path TRUE)
(y_grid_ref 3)
(access_to Sea_Cliffs Sea_Shore))

(LOCATION
(name Mountains)
(text "Mountains")
(x_grid_ref 2)
(y_grid_ref 2)
(path FALSE)
(access_to N_Windy_Track Mountain_Path S_Windy_Track))

(LOCATION
(name Mountain_Path)
(text "Mountain Path")
(x_grid_ref 3)
(y_grid_ref 2)
(path TRUE)
(access_to Mountains Cairn))

(LOCATION
(name Cairn)
(text "Cairn")
(x_grid_ref 4)
(y_grid_ref 2)
(path FALSE)
(access_to Mountain_Path N_Wood_Path Shore_Path S_Wood_Path))

(LOCATION
(name Shore_Path)
(text "Shore Path")
(x_grid_ref 5)
(y_grid_ref 2)
(path TRUE)
(access_to Sea_Shore Cairn))

(LOCATION
(name Sea_Shore)
(text "Sea Shore")
(x_grid_ref 6)
(y_grid_ref 2)
(path FALSE)

```

```

(access_to Shore_Path S_Coastal_Path N_Coastal_Path))

(LOCATION
  (name S_Windy_Track)
  (text "South Windy Track")
  (x_grid_ref 2)
  (y_grid_ref 1)
  (path TRUE)
  (access_to Mountains Bridge))

(LOCATION
  (name S_Wood_Path)
  (text "South Wood Path")
  (x_grid_ref 4)
  (y_grid_ref 1)
  (path TRUE)
  (access_to Cairn Sopron))

(LOCATION
  (name S_Coastal_Path)
  (text "South Coastal Path")
  (x_grid_ref 6)
  (y_grid_ref 1)
  (path TRUE)
  (access_to Sea_Shore Sea_Cove))

(LOCATION
  (name Bridge)
  (text "Bridge")
  (x_grid_ref 2)
  (y_grid_ref 0)
  (path FALSE)
  (access_to S_Windy_Track E_Valley_Path))

(LOCATION
  (name E_Valley_Path)
  (text "East Valley Path")
  (x_grid_ref 3)
  (y_grid_ref 0)
  (path TRUE)
  (access_to Bridge Sopron))

(LOCATION
  (name Sopron)
  (text "Sopron")
  (proper TRUE)
  (x_grid_ref 4)
  (y_grid_ref 0)
  (path FALSE)
  (access_to E_Valley_Path S_Wood_Path Cove_Path))

(LOCATION
  (name Cove_Path)
  (text "Cove Path")
  (x_grid_ref 5)
  (y_grid_ref 0)
  (path TRUE)

```

```

(access_to Sopron Sea_Cove))

(FUNCTIONAL-LOCATION
(name Sea_Cove)
(text "Sea Cove")
(type entry_exit_point)
(x_grid_ref 6)
(y_grid_ref 0)
(path FALSE)
(access_to Cove_Path S_Coastal_Path))

;;;;;;;;;;;;;
;; PEOPLE ;;
;;;;;;;;;;;;;

(TEDDY
(name Hector)
(text "Hector")
(proper TRUE))

(VILLAIN
(name Lord_Speke)
(text "Lord Speke")
(proper TRUE))

(VILLAIN
(name Hughie)
(text "Hughie")
(proper TRUE))

(SPIRIT
(name Hectorina)
(text "Hectorina")
(proper TRUE))

;;;;;;;;;;;;;
;; THINGS;;
;;;;;;;;;;;;;

(ESSENTIAL
(name Chest)
(text "Medicine Chest"))

(PROTECTOR
(name Shield)
(text "Shield"))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; ENGINE INSTANCES ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(LISTER (type locations))

(NARRATOR-COUNTER (value 0)(number_of_things_narrated_so_far 0))

```

```
(TURN-COUNTER (value 0))
```

```
)
```

## 5.2.4 main.clp

```
*** This is the main (default) module.

; How the MAIN module works:

; When '(batch lab)' is called, MAIN is the only module on the
; focus stack.

; setup_animated_map is the first rule to fire, this initialises
; the Java map classes (function initAnimap is in mapcreator.clp)

; collect_locations is the second rule to fire, this collects all
; the LOCATIONS into a single data set, which is later used for
; processing.

; The HERO is then set up at his allocated starting positions,
; whilst all LOCALS and THINGS are set up at a random location
; (a location chosen at random from the location data set mentioned
; earlier)

; Once this is done, focus is passed onto the SYSTEM module to
; start the play cycle.

; Control will not return to MAIN until execution has completed,
; at which point shut_down will terminate execution.

(defrule setup_animated_map
  (declare (salience 15))
  =>
  (initAnimap))

(defrule collect_locations
  (declare (salience 10))
  ?lister <- (LISTER (type locations) (contents ))
  =>
  (collectLocations ?lister)
  (sendLocations ?lister))

(defrule setup_teddy
  (declare (salience 5))
  ?teddy <- (TEDDY
    (name ?hero_name)
    (text ?hero_text)
    (at nil))
  (FUNCTIONAL-LOCATION
    (type entry_exit_point)
    (name ?locationName)
    (x_grid_ref ?x_pos)
    (y_grid_ref ?y_pos))
  =>
  (tellNarrator at ?hero_name ?locationName)
  (mapSetHero ?x_pos ?y_pos ?hero_text)
  (modify ?teddy (at ?locationName)))

(defrule setup_locals
```

```

(declare (salience 5))
?local <- (LOCAL
  (name ?local_name)
  (text ?local_text)
  (at nil))
(LISTER (type locations) (contents $?locations))
=>
(bind ?random_location (getRandom $?locations))
(tellNarrator at ?local_name ?random_location)
(mapSetLocal ?random_location ?local_text)
(modify ?local (at ?random_location)))

(defrule setup_things
(declare (salience 5))
?thing <- (THING
  (name ?thing_name)
  (text ?thing_text)
  (at nil)
  (carried_by nil))
(LISTER (type locations) (contents $?locations))
=>
(bind ?random_location (getRandom $?locations))
(tellNarrator at ?thing_name ?random_location)
(mapSetThing ?random_location ?thing_text)
(modify ?thing (at ?random_location)))

(defrule start_up
=>
(focus SYSTEM))

(defrule shut_down
(not (NARRATION))
=>
(clear-focus-stack))

(deffunction collectLocations (?lister)
; recursively add all LOCATIONs onto the end of the ?lister, by
; means of a java.util.Iterator
; ?lister needs to be a slot named 'contents' which is empty at
; the start of the Iteration.
(bind ?locationsIterator (run-query getLocation))
(while (?locationsIterator hasNext)
  (bind ?token (call ?locationsIterator next))
  (bind ?location_fact (call ?token fact 1))
  (modify ?lister (contents (addToList (fact-slot-value ?location_fact name) ?lister))))))

(deffunction addToList (?element ?list)
; if the ?list is empty, then adds the ?element to the ?list,
; else adds the ?element onto the end of the ?list
(if (= (length$ (fact-slot-value ?list contents)) 0)
  then (return ?element)
  else (return (insert$
    (fact-slot-value ?list contents)
    (+ 1 (length$ (fact-slot-value ?list contents))) ?element))))

(defquery getLocation

```

```

; returns a java.util.Iterator object containing all LOCATIONS
; and their grid_refs
  (LOCATION (name ?name) (x_grid_ref ?x) (y_grid_ref ?y)))

(defun getRandom ($?list)
; return an element at random from a list with at least one element
  (return (nth$ (+ 1 (mod (random) (length$ $?list))) $?list)))

(defun tellNarrator ($?args)
; after accessing variables, first prints out a rule trace in upper case,
; then creates a fact of type NARRATION, with slots determined by the
; call parameters, and increments the NARRATOR-COUNTER
; Get the correct arguments
  (bind ?predicate (nth$ 1 $?args))
  (bind ?arg1 (nth$ 2 $?args))
  (bind ?arg2 (nth$ 3 $?args))

; Get the narrator counter
  (bind ?narrator_iterator (run-query getNarratorCounter))
  (bind ?token (call ?narrator_iterator next))
  (bind ?narrator_counter (call ?token fact 1))

; Print narration
  (printout t (upcase ?predicate) " " (toUpper ?arg1) " " (toUpper ?arg2) crlf)

; Update counter
  (bind ?count (+ 1 (fact-slot-value ?narrator_counter value)))
  (modify ?narrator_counter (value ?count))

; Actually "tell the narrator" what to say
  (assert (NARRATION
    (predicate ?predicate)
    (arg1 ?arg1)
    (arg2 ?arg2)
    (index ?count)))
)

(defun toUpper (?string)
; Returns the given string in upper case, or empty string for nil
  (if (eq ?string nil) then (return ""))
  (return (upcase ?string)))

(defquery getNarratorCounter
; Returns a java.util.Iterator of all 1 of the narrator counters
; for the tellNarrator function
  (NARRATOR-COUNTER))

(defquery getAccessibleLocations
; returns a java.util.Iterator of all locations accessible by the
; selected location
; (i.e. the input parameter location) that can be used to obtain
; information about those locations.
; The second item (fact) of each match will be the location that
; is in the access_to slot.
  (declare (variables ?current_location))
  (LOCATION

```

```
(name ?current_location)
(access_to $?access))
(LOCATION
(name ?return_location&:(neq FALSE (member$ ?return_location $?access))))
(LOCATION
(name ?return_location&~?current_location))
```

### 5.2.5 system.clp

```
; How the SYSTEM module works:

; System is the lynchpin of Hector's World and controls the two
; main abstract entities (the story creator and the story teller)
; in a turn-based style.

; In essence, it lets the story teller say whatever it has to say
; first. Once it (the story teller) has finished, SYSTEM lets the
; story creator 'play', after it has reactivated (woken up) all the
; players and incremented the turn-counter. When that (the story
; creator) has finished, SYSTEM lets the story teller say whatever
; the story creator has done in the last turn, and so on.

; This cycle ends once SYSTEM notices that the story creator has
; finished (i.e. the TEDDY has escaped), and the story teller has
; nothing else to say, at which point control returns to MAIN.
; This is enforced by only letting the story creator play if the
; hero has not yet escaped.
```

```
(defmodule SYSTEM)

(defrule tell_story
  (declare (salience 3))
  (not (not_finished))
  =>
  (focus NARRATOR))

(defrule check_teddy
  (declare (salience 2))
  (not (TEDDY (to_go_to freedom)))
  (not (not_finished))
  =>
  (assert (not_finished)))

(defrule reactivate
  (declare (salience 1))
  ?person <- (PERSON (active FALSE))
  (not_finished)
  =>
  (modify ?person (active TRUE)))

(defrule go_on
  ?not_finished <- (not_finished)
  ?turn_counter <- (TURN-COUNTER (value ?turns))
  =>
  (retract ?not_finished)
  (modify ?turn_counter (value (+ ?turns 1)))
  (bind ?sentence (str-cat "TURN " (+ ?turns 1) ": "))
  (printout t ?sentence crlf)
  (mapPrintSentence ?sentence)
  (focus WORLD))
```

## 5.2.6 world.clp

```
; How the WORLD module works:

; WORLD represents the abstraction of the story creator.
; It plays the adventure.

; When the module is called (i.e. it is at the top of the
; focus stack), it will let first the LOCALs play, and then
; the HERO. This order is because the HERO will often react
; to what the LOCALs do. Once each LOCAL and then the HERO
; has played, this module will have nothing else to do, and
; so control will return to the calling module (SYSTEM).

(defmodule WORLD)

(defrule local_turn
  (declare (salience 1))
  (LOCAL
   (active TRUE))
  =>
  (focus LOCAL))

(defrule hero_turn
  (TEDDY
   (active TRUE))
  =>
  (focus HERO-MAIN)
  (mapAllMovesGiven))
```

### 5.2.7 local.clp

```
; How the LOCAL module works:

; LOCAL does two things:

; First, it moves each LOCAL (in turn).

; Second, it checks to see if any of the LOCALs run into
; the TEDDY. If this happens, the LOCALs 'find' the TEDDY,
; and nothing else will happen at this point. Any interaction
; between the TEDDY and any LOCAL will be dealt with by the
; HERO-ACTION module.

; Once all LOCALs have moved, and have been checked (to see
; if they find the TEDDY), control returns (to WORLD).
```

```
(defmodule LOCAL)

(defrule move_local
  (declare (salience 1))
  ?local <- (LOCAL
    (text ?local_text)
    (active TRUE)
    (at ?location))
  (LOCATION
    (name ?location)
    (access_to $?locations_to_choose_from))
  =>
  (bind ?new_location (getRandom $?locations_to_choose_from))
  (mapMoveAllPeople ?local_text ?new_location)
  (modify ?local (at ?new_location) (active FALSE)))

(defrule check_local
  (LOCAL
    (name ?local_name)
    (at ?location))
  (TEDDY
    (name ?teddy_name)
    (at ?location))
  (not (NARRATION (predicate finds)))
  =>
  (tellNarrator finds ?local_name ?teddy_name))
```

### 5.2.8 hero-main.clp

```
; How the HERO-MAIN module works:  
  
; This module plays the HERO. First of all it will take care  
; of any events that are outside of his control (meetings with  
; LOCALs, etc) by focussing on the HERO-MEETING module. Then, if  
; the HERO is still able to, it will let him play (by focussing  
; on the HERO-ACTION module).
```

```
(defmodule HERO-MAIN)
```

```
(defrule meet_local  
  (declare (salience 1))  
  (TEDDY  
    (at ?location)  
    (active TRUE))  
  (LOCAL  
    (at ?location))  
  =>  
  (focus HERO-MEETING))
```

```
(defrule play_hero  
  (TEDDY  
    (active TRUE))  
  =>  
  (focus HERO-ACTION))
```

### 5.2.9 hero-meeting.clp

```
; How the HERO-MEETING module works:

; HERO-MEETING deals with situations when the TEDDY stumbles across
; one of the LOCALS who inhabit the island. He has little control
; over these situations.

; If the TEDDY stumbles across HECTORINA (the spirit), then focus
; is passed to the HERO-SPIRIT module to determine how she helps him.

; If the TEDDY is still active (either because there was no spirit
; about, or because the spirit has not done anything which has
; inactivated the TEDDY), and the TEDDY stumbles across a VILLAIN,
; then focus is passed to HERO-VILLAIN to resolve the encounter. If
; there is more than one VILLAIN then the TEDDY will need to face each
; VILLAIN in turn (unless one of the VILLAINS forces the TEDDY to do
; something which inactivates him first).

; Once there are no more events to deal with (either because the TEDDY
; is no longer active or because all events have already been dealt
; with), control will return (to HERO-MAIN).
```

```
(defmodule HERO-MEETING)

(defrule meet_spirit
  (declare (salience 1))
  (TEDDY
   (name ?teddy_name)
   (at ?location)
   (active TRUE)
   (to_go_to ?to_go_to_location))
  (LOCATION
   (name ?to_go_to_location))
  (SPIRIT
   (name ?spirit_name)
   (at ?location))
  (not (NARRATION (predicate helps)))
  =>
  (tellNarrator sees ?teddy_name ?spirit_name)
  (tellNarrator helps ?spirit_name ?teddy_name)
  (focus HERO-SPIRIT))

(defrule meet_villain
  (TEDDY
   (name ?teddy_name)
   (at ?location)
   (active TRUE))
  (VILLAIN
   (name ?villain_name)
   (at ?location))
  (not (NARRATION (predicate challenges)))
  =>
  (tellNarrator sees ?teddy_name ?villain_name)
  (tellNarrator challenges ?villain_name ?teddy_name)
  (focus HERO-VILLAIN))
```

### 5.2.10 hero-spirit.clp

```
; How the HERO-SPIRIT module works:  
  
; Right now HERO-SPIRIT does nothing.  
  
(defmodule HERO-SPIRIT)
```

### 5.2.11 hero-villain.clp

```
; How the HERO-VILLAIN module works:

; When the TEDDY encounters a VILLAIN, one of two things can happen:

; If the TEDDY has carries a PROTECTOR, he will stand against the VILLAIN
; (nothing happens, and the encounter is resolved).

; If the TEDDY is not holding a PROTECTOR, he will be forced to flee in a
; random direction. This inactivates him.

; Once either of these events have happened, control will return
; (to HERO-MEETING).
```

```
(defmodule HERO-VILLAIN)

(defrule stand
  (TEDDY
   (name ?teddy_name)
   (at ?location))
  (VILLAIN
   (name ?villain_name)
   (at ?location))
  (PROTECTOR
   (name ?protector_name)
   (carried_by ?teddy_name))
  (not (NARRATION (predicate stands)))
  =>
  (tellNarrator contrast)
  (tellNarrator holds ?teddy_name ?protector_name)
  (tellNarrator conseq)
  (tellNarrator stands ?teddy_name ?villain_name)
  (mapSetTeddyStanding))

(defrule flee
  ?teddy <- (TEDDY
   (name ?teddy_name)
   (text ?teddy_text)
   (at ?location))
  (LOCATION
   (name ?location) (access_to $?possible_flee_locations))
  (not (PROTECTOR
   (name ?protector_name)
   (carried_by ?teddy_name)))
  (not (NARRATION (predicate flees)))
  =>
  (bind ?flee_location (getRandom $?possible_flee_locations))
  (tellNarrator conseq)
  (tellNarrator flees ?teddy_name ?flee_location)
  (mapMoveAllPeople ?teddy_text ?flee_location)
  (mapSetTeddyFleeing)
  (modify ?teddy (at ?flee_location) (active FALSE)))
```

### 5.2.12 hero-action.clp

```
; How the HERO-ACTION module works:

; By default, the first thing that the TEDDY will try to do is,
; if there is a THING that he sees, he will pick it up. This will
; use his turn up (inactivate him). He will then have nothing
; else to do and control will return (to HERO-MAIN).

; Otherwise if he has no goal (if he doesn't know what he wants
; to do), the TEDDY will call HERO-GOAL to determine that goal,
; then continue.

; If the TEDDY has a goal, but he has already achieved that goal,
; then that goal will be eliminated (i.e. he will no longer have a
; goal). This will activate and then immediately fire the set_goal
; rule, since he has no goal.

; Finally, if the TEDDY has not used up his turn to pick up a THING,
; he knows where he wants to go (and he is not already there), then
; he will move towards that goal. At this point he will have used
; up his turn (the TEDDY will be inactivated), and the focus will
; return (to HERO-MAIN)
```

```
(defmodule HERO-ACTION)

(defrule see_thing
  (declare (salience 4))
  ?teddy <- (TEDDY
    (name ?teddy_name)
    (at ?location)
    (active TRUE))
  ?thing <- (THING
    (name ?thing_name)
    (at ?location))
  (not (NARRATION (predicate sees)))
  =>
  (tellNarrator sees ?teddy_name ?thing_name))

(defrule take_thing
  (declare (salience 3))
  ?teddy <- (TEDDY
    (name ?teddy_name)
    (at ?location)
    (active TRUE))
  ?thing <- (THING
    (name ?thing_name)
    (text ?thing_text)
    (at ?location))
  (not (NARRATION (predicate takes)))
  =>
  (tellNarrator takes ?teddy_name ?thing_name)
  (mapBagThing ?thing_text)
  (modify ?thing (carried_by ?teddy_name) (at nil))
  (modify ?teddy (active FALSE)))
```

```

(defrule set_goal
  (declare (salience 2))
  (TEDDY
   (active TRUE)
   (to_go_to nil))
  =>
  (focus HERO-GOAL))

(defrule check_goal
  (declare (salience 1))
  ?teddy <- (TEDDY
   (at ?location)
   (active TRUE)
   (to_go_to ?location))
  =>
  (modify ?teddy (to_go_to nil)))

(defrule move
  ?teddy <- (TEDDY
   (name ?teddy_name)
   (text ?teddy_text)
   (active TRUE)
   (at ?current_location)
   (to_go_to ?goal_location))
  (LOCATION
   (name ?goal_location)
   (x_grid_ref ?goal_x_ref)
   (y_grid_ref ?goal_y_ref))
  (not (NARRATION (predicate moves)))
  =>
  (bind ?location_to_move_to
   (getLocationInDirectionOfGoal ?current_location ?goal_x_ref ?goal_y_ref))
  (tellNarrator moves ?teddy_name ?location_to_move_to)
  (mapMoveAllPeople ?teddy_text ?location_to_move_to)
  (mapSetTeddyNormal)
  (modify ?teddy (at ?location_to_move_to) (active FALSE)))

(deffunction getLocationInDirectionOfGoal
  (?current_location ?goal_x_ref ?goal_y_ref)

;PRECONDITIONS: The current location is != the goal location,
; and it has at least one accessTo location, which it can move to.
;POSTCONDITIONS: base case: if the goal can be reached in 1 turn,
; then move to the goal,
; else do a depth-2 greedy search to find the first location
; which brings you closer to your goal, and pick it
; else (failsafe case) just pick the last location looked at.
; This 'give up' move will never happen in the basic hector's world
; scenario, but is included to avoid potential deadlocks in the case
; where the hero and his goal are directly opposite, yet there is no
; direct path between them. This can only happen when there is more
; than one 'essential' or if the map structure is changed.

; set up the iteration
(bind ?selected_locations_iterator

```

```

(run-query getAccessibleLocations ?current_location))
(while (?selected_locations_iterator hasNext)
  (bind ?token (call ?selected_locations_iterator next))

  (bind ?current_location_fact (call ?token fact 1))
  (bind ?current_x_coord (fact-slot-value ?current_location_fact x_grid_ref))
  (bind ?current_y_coord (fact-slot-value ?current_location_fact y_grid_ref))

  (bind ?accessible_location_fact (call ?token fact 2))
  (bind ?accessible_x_coord (fact-slot-value ?accessible_location_fact x_grid_ref))
  (bind ?accessible_y_coord (fact-slot-value ?accessible_location_fact y_grid_ref))
  (bind ?accessible_location_name (fact-slot-value ?accessible_location_fact name))

;base case
(if
  (and
    (eq ?accessible_x_coord ?goal_x_ref)
    (eq ?accessible_y_coord ?goal_y_ref)
  )
  then
    (return ?accessible_location_name)
  )

;depth 2 greedy search
(bind ?depth_2_iterator (run-query getAccessibleLocations ?accessible_location_name))
(while (?depth_2_iterator hasNext)
  (bind ?token_2 (call ?depth_2_iterator next))
  (bind ?depth_2_location_fact (call ?token_2 fact 2))
  (bind ?depth_2_x_coord (fact-slot-value ?depth_2_location_fact x_grid_ref))
  (bind ?depth_2_y_coord (fact-slot-value ?depth_2_location_fact y_grid_ref))
  (if
    (<
      (+
        (abs (- ?goal_x_ref ?depth_2_x_coord))
        (abs (- ?goal_y_ref ?depth_2_y_coord))
      )
      (+
        (abs (- ?goal_x_ref ?current_x_coord))
        (abs (- ?goal_y_ref ?current_y_coord))
      )
    )
    then (return ?accessible_location_name)
  )
  )
)

;default case
(return ?accessible_location_name)
)

```

### 5.2.13 narrator.clp

```
; How the NARRATOR module works:

; This module represents the story teller. If there is something
; (a NARRATION) to say, then focus will be passed to the LEXITEMS
; module to determine what to say next.

; When there are no more NARRATIONS to deal with, the NARRATOR
; module will close things off by printing out the final sentence
; followed by a dashed line to indicated that it has finished telling
; everything it has to say, this turn (this dashed line indicates the
; start/end of each turn on screen).

; Once everything has been said and done, all counters are reset
; for the start of a new cycle, and control returns to SYSTEM.
```

```
(defmodule NARRATOR)

(defrule lexitems
  (NARRATION)
  ?narratorCounter <- (NARRATOR-COUNTER (number_of_things_narrated_so_far ?counter))
  =>
  (modify ?narratorCounter (number_of_things_narrated_so_far (+ ?counter 1)))
  (focus LEXITEMS))

(defrule last_sentence
  ?sentence <- (SENTENCE (text ?text))
  ?narrator-counter <- (NARRATOR-COUNTER
    (number_of_things_narrated_so_far ?narrated) (value ?toNarrate))
  (test (>= ?narrated ?toNarrate))
  =>
  (printSentence ?text)
  (printout t crlf "-----" crlf)
  (mapPrintLine)
  (retract ?sentence)
  (resetNarratorCounter ?narrator-counter))

(deffunction setDeterminer (?text ?flag)
  (if ?flag
    then return ?text
    else return (str-cat "the " ?text)))
; if the flag = TRUE, return "text", else return "the text"

(deffunction printSentence (?text)
; print the sentence contained in ?text followed by a full stop
  (printout t ?text ". " crlf)
  (mapPrintSentence ?text))

(deffunction resetNarratorCounter (?counter)
  (modify ?counter (value 0)(number_of_things_narrated_so_far 0)))
```

### 5.2.14 lexitems.clp

```
; How the LEXITEMS module works:

; This module translates the information (the NARRATIONS) given
; to it by the story teller into pieces that can be used to form
; natural language (i.e. English) sentences.

; It only deals with the current NARRATION (this is controlled by a
; counter), which it matches with one of the rules to produce the
; components (nouns, verbs, etc) that make up sentences.

; This done, it will retract (delete) the NARRATION, as this is no
; longer necessary.

; Then, LEXITEMS will pass control over to LEXRULES, which will
; use the pieces produced by LEXITEMS and put them together.

; Once LEXRULES has finished, control will automatically return to
; LEXITEMS, which will, in turn, return to NARRATOR. If there are more
; NARRATIONS to deal with, then NARRATOR will increment the relevant ;
; counters and call LEXITEMS again.
```

```
(defmodule LEXITEMS)
; rules are in alphabetical order, except for the control rule,
; which is at the start, and the catch-all rule at the end.
```

```
(defrule continue
  (NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
  (not (NARRATION (index ?i)))
  =>
  (focus LEXRULES))
```

```
(defrule at
  (NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
  ?narration <- (NARRATION
    (predicate at)
    (arg1 ?arg1)
    (arg2 ?arg2)
    (index ?i))
  (MOBILE
    (name ?arg1))
  (LOCATION
    (name ?arg2))
  =>
  (bind ?text "is at")
  (createLexitems ?arg1 ?arg2 ?text)
  (retract ?narration))
```

```
(defrule cause
  (NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
  ?narration <- (NARRATION
    (predicate cause)
    (arg1 nil)
    (arg2 nil))
```

```

    (index ?i))
=>
(bind ?text "because")
(assert (LINK (text ?text)))
(retract ?narration))

(defrule contrast
(NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
?narration <- (NARRATION
  (predicate contrast)
  (arg1 nil)
  (arg2 nil)
  (index ?i))
=>
(bind ?text "but")
(assert (LINK (text ?text)))
(retract ?narration))

(defrule conseq
(NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
?narration <- (NARRATION
  (predicate conseq)
  (arg1 nil)
  (arg2 nil)
  (index ?i))
=>
(bind ?text "so")
(assert (LINK (text ?text)))
(retract ?narration))

(defrule challenges
(NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
?narration <- (NARRATION
  (predicate challenges)
  (arg1 ?challengerName)
  (arg2 ?challengedName)
  (index ?i))
(PERSON
  (name ?challengerName))
(PERSON
  (name ?challengedName))
=>
(bind ?text "challenges")
(createLexitem ?challengerName ?challengedName ?text)
(retract ?narration))

(defrule escapes
(NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
?narration <- (NARRATION
  (predicate escapes)
  (arg1 ?escapee)
  (index ?i))
(PERSON
  (name ?escapee))
=>
(bind ?text "escapes from the island")
(createSimpleLexitem ?escapee ?text)

```

```

(retract ?narration))

(defrule finds
(NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
?narration <- (NARRATION
  (predicate finds)
  (arg1 ?finderName)
  (arg2 ?foundName)
  (index ?i))
(PERSON
  (name ?finderName))
(PERSON
  (name ?foundName))
=>
(bind ?text "finds")
(createLexitems ?finderName ?foundName ?text)
(retract ?narration))

(defrule flees
(NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
?narration <- (NARRATION
  (predicate flees)
  (arg1 ?fleerName)
  (arg2 ?locationFleingTo)
  (index ?i))
(PERSON
  (name ?fleerName))
(LOCATION
  (name ?locationFleingTo))
=>
(bind ?text "flees to")
(createLexitems ?fleerName ?locationFleingTo ?text)
(retract ?narration))

(defrule flies
(NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
?narration <- (NARRATION
  (predicate flies)
  (arg1 ?flierName)
  (arg2 ?locationFlyingTo)
  (index ?i))
(PERSON
  (name ?flierName))
(LOCATION
  (name ?locationFlyingTo))
=>
(bind ?text "is flown to")
(createLexitems ?flierName ?locationFlyingTo ?text)
(retract ?narration))

(defrule helps
(NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
?narration <- (NARRATION
  (predicate helps)
  (arg1 ?helperName)
  (arg2 ?helpedName)

```

```

    (index ?i))
(PERSON
  (name ?helperName))
(PERSON
  (name ?helpedName))
=>
(bind ?text "helps")
(createLexitems ?helperName ?helpedName ?text)
(retract ?narration))

(defrule holds
(NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
?narration <- (NARRATION
  (predicate holds)
  (arg1 ?holderName)
  (arg2 ?heldName)
  (index ?i))
(PERSON
  (name ?holderName))
(THING
  (name ?heldName))
=>
(bind ?text "is holding")
(createLexitems ?holderName ?heldName ?text)
(retract ?narration))

(defrule moves
(NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
?narration <- (NARRATION
  (predicate moves)
  (arg1 ?moverName)
  (arg2 ?locationMovingTo)
  (index ?i))
(PERSON
  (name ?moverName))
(LOCATION
  (name ?locationMovingTo))
=>
(bind ?text "moves to")
(createLexitems ?moverName ?locationMovingTo ?text)
(retract ?narration))

(defrule seeks
(NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
?narration <- (NARRATION
  (predicate seeks)
  (arg1 ?seekerName)
  (arg2 ?thingBeingSought)
  (index ?i))
(PERSON
  (name ?seekerName))
(ENTITY
  (name ?thingBeingSought))
=>
(bind ?text "seeks")
(createLexitems ?seekerName ?thingBeingSought ?text)
(retract ?narration))

```

```

(defrule sees
  (NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
  ?narration <- (NARRATION
    (predicate sees)
    (arg1 ?seerName)
    (arg2 ?foundName)
    (index ?i))
  (PERSON
    (name ?seerName))
  (MOBILE
    (name ?foundName))
  =>
  (bind ?text "sees")
  (createLexitems ?seerName ?foundName ?text)
  (retract ?narration))

```

```

(defrule stands
  (NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
  ?narration <- (NARRATION
    (predicate stands)
    (arg1 ?standerName)
    (arg2 ?standsAgainstName)
    (index ?i))
  (PERSON
    (name ?standerName))
  (PERSON
    (name ?standsAgainstName))
  =>
  (bind ?text "stands his ground against")
  (createLexitems ?standerName ?standsAgainstName ?text)
  (retract ?narration))

```

```

(defrule takes
  (NARRATOR-COUNTER (number_of_things_narrated_so_far ?i))
  ?narration <- (NARRATION
    (predicate takes)
    (arg1 ?takerName)
    (arg2 ?thingName)
    (index ?i))
  (PERSON
    (name ?takerName))
  (THING
    (name ?thingName))
  =>
  (bind ?text "takes")
  (createLexitems ?takerName ?thingName ?text)
  (retract ?narration))

```

```

(defrule no_match
; Catch-all when NARRATION not matched
  (declare (salience -5))
  ?narrator_counter <- (NARRATOR-COUNTER
    (number_of_things_narrated_so_far ?n))
  ?narration <- (NARRATION (index ?i))
  (test (eq ?n ?i))

```

```
=>
(retract ?narration)
(modify ?narrator_counter (number_of_things_narrated_so_far (+ ?n 1)))

(defun createLexitems (?arg1 ?arg2 ?text)
  (assert (NOUN (source ?arg1) (role agent)))
  (assert (NOUN (source ?arg2) (role object)))
  (assert (VERB (text ?text))))

(defun createSimpleLexitem (?arg1 ?text)
  (assert (NOUN (source ?arg1) (role agent)))
  (assert (VERB (text ?text))))
```

### 5.2.15 lexrules.clp

```
; How the LEXRULES module works:

; The first thing that LEXRULES will do will be to print off any
; sentences that have been produced previously.

; Then, LEXRULES is used to assemble the basic pieces (nouns, verbs,
; etc) produced by LEXITEMS and put them together into one sentence.
; It will do this by:

; First, it will check (and change as necessary) to see if "the"
; needs to be put in front of any nouns ("the shield" for example,
; instead of simple "shield").

; Then, it will assemble any object noun (a NOUN which is the object
; of a sentence) and verb pair into a VERBPHRASE. It will then convert
; this VERBPHRASE into a sentence by combining it with a NOUN of type
; agent.

; If the sentence has no object (no NOUN of type object) the verb
; will be combined directly with the agent (NOUN of type agent) to
; form a simple_sentence

; If there is a LINK (but, so, etc) then the sentence will be be
; combined with the link to form the head of a sentence (which will
; later be combined with the next sentence to make a full sentence).

; Once all of the above is done, control will return (to LEXITEMS
; and then directly onto NARRATOR). The sentence that was formed
; here will be printed off the next time LEXRULES is called (if it
; is a normal sentence) or else it will be combined with the next
; sentence (if it is only the head of a sentence). In the case where
; there is no next sentence (i.e. LEXRULES will not be called again
; this turn), then the NARRATOR will take care of printing out the
; sentence.
```

```
(defmodule LEXRULES)
```

```
(defrule check_determiner
  (declare (salience 10))
  ?noun <- (NOUN
    (source ?source)
    (determined FALSE))
  (ENTITY
    (name ?source)
    (text ?text)
    (proper ?proper))
  =>
  (bind ?determined_noun (setDeterminer ?text ?proper))
  (modify ?noun (text ?determined_noun) (determined TRUE)))
```

```
(defrule print_previous_sentence
  (declare (salience 10))
  ?sentence <- (SENTENCE
    (role normal)
```

```

    (text ?text))
(NOUN)
=>
(printSentence ?text)
(retract ?sentence))

(defrule create_verb_phrase
(declare (salience 5))
?verb <- (VERB
    (text ?very_text))
?noun <- (NOUN
    (text ?noun_text)
    (role object))
=>
(bind ?verb_phrase_text (str-cat ?very_text " " ?noun_text))
(assert (VERBPHRASE (text ?verb_phrase_text)))
(retract ?verb ?noun))

(defrule create_sentence
(declare (salience 5))
?verb_phrase <- (VERBPHRASE
    (text ?verb_phrase_text))
?noun <- (NOUN
    (text ?noun_text)
    (role agent))
=>
(bind ?sentence_text (str-cat ?noun_text " " ?verb_phrase_text))
(assert (SENTENCE (text ?sentence_text) (role normal)))
(retract ?verb_phrase ?noun))

(defrule create_simple_sentence
?verb <- (VERB
    (text ?very_text))
?noun <- (NOUN
    (text ?noun_text)
    (role agent))
=>
(bind ?sentence_text (str-cat ?noun_text " " ?very_text))
(assert (SENTENCE (text ?sentence_text) (role normal)))
(retract ?verb ?noun))

(defrule make_normal_sentence_into_head_of_sentence
?sentence <- (SENTENCE
    (role normal))
(LINK)
=>
(modify ?sentence (role head)))

(defrule make_full_sentence
?head_sentence <- (SENTENCE
    (text ?head_text)
    (role head))
?normal_sentence <- (SENTENCE
    (text ?normal_text)
    (role normal))
?link <- (LINK
    (text ?link_text))

```

```
=>
(bind ?full_text (str-cat ?head_text " " ?link_text " " ?normal_text))
(assert (SENTENCE (text ?full_text) (role normal)))
(retract ?head_sentence ?normal_sentence ?link))

(defun setDeterminer (?text ?flag)
  (if ?flag
      then (return ?text)
      else (return (str-cat "the " ?text))))
```

### 5.2.16 buggy1.clp

```
(defrule help_hero_find_essential
  ?teddy <- (TEDDY
    (name ?teddy_nam)
    (text ?teddy_text)
    (to_go_to ?location)
    (active TRUE))
  (ESSENTIAL
    (name ?essential-name)
    (at ?location)
    (carried_by nil))
  (LOCATION
    (name ?location))
  (not (NARRATION (predicate at)))
  =>
  (tellNarrator conseq)
  (tellNarrator flies ?teddy_name ?location)
  (tellNarrator cause)
  (tellNarrator at ?essential_name ?location)
  (mapMoveAllPeople ?teddy_text ?location)
  (mapSetTeddyFlying)
  (modify ?teddy (at ?location) (active FALSE)))
```

### 5.2.17 buggy2.clp

```
(defrule to_the_sea_cove
  ?teddy <- (TEDDY
    (name ?teddy_name)
    (text ?teddy_text)
    (active TRUE))
  (LOCATION
    (name ?cove)
    (text "Sea Cove))
  (not (ESSENTIAL
    (name essential_name)
    (carried_by nil)))
  (not (NARRATION (predicate flies)))
  =>
  (tellNarrator conseq)
  (tellNarrator flies ?teddy_name ?cove)
  (mapMoveAllPeople ?teddy_text ?cove)
  (mapSetTeddyFlying)
  (modify ?teddy (at ?cove) (active FALSE)))
```

## Credits

The original KnowledgeWorks “Hector’s World” lab exercises and manual were conceived, written, and refined by Joe Bullock, Crispin Miller, Mark Quinn, Alan Rector, and Mary McGee Wood

With support and inspiration from all the students on CS2422 1995 and 1996, especially Tim Dinsdale, Chris Sharples, and Gareth Williams.

Lab ported to CLIPS by John Parkinson; exercises and manual re-written for CLIPS by John Parkinson and Mary McGee Wood.

The lab was later converted to JESS by Pol Rakower. The current lab exercises and manual were written by Pol Rakower and Mary McGee Wood, revised by Phil Reed, and tested by Stavros Antoniou, Aliya Aziz, and Phil Reed. The animated-map was designed by Aliya Aziz.

Cover illustration by Jon Hulme, colour by Phil Reed.