How to speak Genesese

Patrick H. Winston

Revision of June 17, 2017

Genesese is the subset of English understood by Genesis. *Understood* means that the English is parsable by Boris Katz's START parser, which produces a sort of semantic net, and a translator in Genesis that translates the semantic net into Genesis's inner language, *Innerese*. All knowledge in Genesis is produced via Genesese-to-Innerese parsing and translation.

Genesese is also the subset of English generated by Genesis. *Generated* means that the Innerese is translated into a form needed by START's generator, which then produces Genesese.

Basics

The best way to understand what you can say in Genesese is to look at a simple sample file containing all the sentences needed to understand a simple story wholly contained in one file:

```
Start experiment.
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
Macbeth murders duncan.
The end.
```

The result appears in the elaboration graph shown in figure 1:

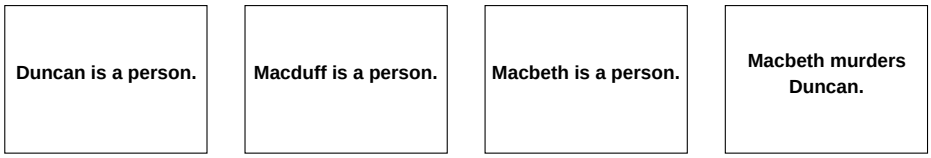

Figure 1: A story with three people and one action.

All that you see, of course, are the statements in the story. The line

```
Start experiment.
```

is an idiom that tells Genesis to clear all memory out of the story processors in preparation for reading a new story. The line

```
Start story titled "Tragedy of Macbeth".
```

is an idiom that tells Genesis that a story is about to be told and the name of the story. The line

```
The end.
```

is an idiom that tells Genesis that the story is concluded.

To read such a file, you click on the Read button and navigate to your file, as shown in figure 2:

Figure 2: Story reading starts with a click on the Read button click.

Comments

Naturally, you will want to comment your work, just as you would code. You use the Java conventions, with a double slash, `//`, introducing a single line comment, and matched `/*` and `*/` pairs bracketing multiline comments:

```
// The idiom for resetting story processor memory:
Start experiment.
/*
A sample story, starting with an instance of the "Start story titled..."
idiom and ending with an instance of the  "The end." idiom.
*/
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
Macduff murders Duncan.
The end.
```

At the moment, you cannot nest `/*` and `*/` pairs.

Rules

Prediction rules

The story becomes more interesting when you add common sense prediction rules. Here, Genesis is told that if a person kills someone, that other person becomes dead:

```
Start experiment.
xx, yy, and zz are persons.
// A prediction rule:
If xx kills yy then yy becomes dead.
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
Macduff kills Macbeth.
The end.
```

Now in the elaboration graph, in yellow, you see the consequent event, and that consequent is connected to its antecedent by a line, as shown in figure 3.

Common sense rules are generally of the form `If ...  then ....`, but you can also use an alternate form with the word `because`:

```
yy becomes dead because xx kills yy.
```

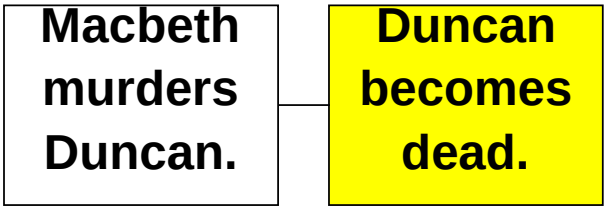

Figure 3: Prediction rules establish connections between one or more antecedents and one consequent.

Explanation rules

Genesis uses explanation rules whenever an event occurs that would otherwise lack an explanation. Consider this version:

```
Start experiment.
xx, yy, and zz are persons.
// A prediction rule:
If xx kills yy then yy becomes dead.
// An explanation rule; note use of "may":
If xx angers yy, then yy may kill xx.
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
Macbeth angers Macduff.
Macduff kills Macbeth.
The end.
```

The explanation rule indicates that if there is no other explanation for yy killing xx, and there is an assertion already that xx angers yy, then there is an assumed causal connection.

The reason for explanation rules, marked with *may*, is that the antecedent, in this example, *xx angers yy*, does not always lead to the consequent, *yy kills xx*. We humans always seek explanations, however, so the rule is used if it supplies an explanation and no other explanation is given.

The result appears in the elaboration graph shown in figure 4:

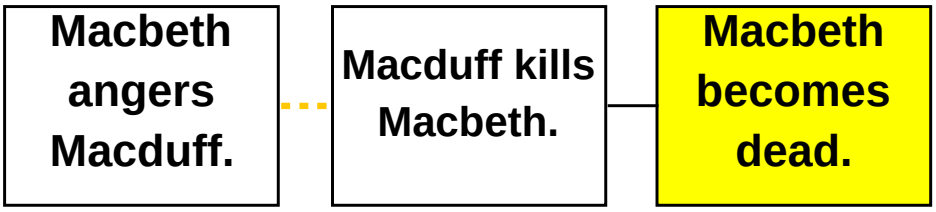

Figure 4: An explanation rule joins a action to a previous action in the absence of any other explanation.

Note that explanation-rule connections are shown in orange and dotted, rather than solid black.

Note also that explanation rules can be, like inference rules, expressed with *because*:

```
xx may murder yy because xx wants to murder yy.
```

Presumption rules

Genesis uses presumption rules whenever there is no other explanation for an action and no explanation rule provides one. Explanation rules require their antecedents to be in a story; presumption rules put antecedents into the story, as in the following example:

```
Start experiment.
xx, yy, and zz are persons.
// A prediction rule:
If xx kills yy then yy becomes dead.
// An explanation rule; note use of "may":
If xx angers yy, then yy may kill xx.
// A presumption rule; note use of "can be"
If xx is foolish, xx can be greedy.
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
Macbeth is greedy.
The end.
```

The result appears in the elaboration graph shown in figure 5:

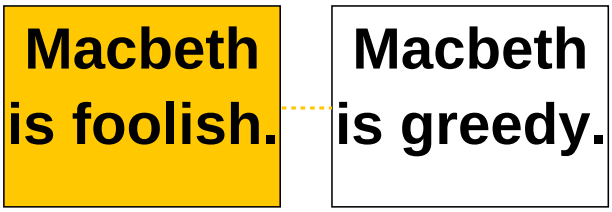

Figure 5: An presumption rule joins an action to an imagined cause in the absence of any other explanation.

Abduction rules

Sometimes we assume a cause when a consequence is stated, thus doing abductive inference. Thus, abduction is something like presumption, but happens no matter whether there is an alternative explanation or not. For example, Genesis can conclude that if someone murders someone else, that person must be insane:

```
Start experiment.
xx, yy, and zz are persons.
// A prediction rule:
If xx kills yy then yy becomes dead.
// An explanation rule; note use of "may":
If xx angers yy, then yy may kill xx.
// A presumption rule; note use of "can be"
If xx is foolish, xx can be greedy.
// An abduction rule; note use of "must be"
If xx kills yy, xx must be insane.
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
Macbeth murders Duncan.
The end.
```

The result is shown in the elaboration graph shown in figure 6:

Enablement rules

Sometimes we assume a condition must hold because a certain type of action occurred. For example, Genesis can conclude that if someone stabs someone else, that person have a knife.

```
Start experiment.
```

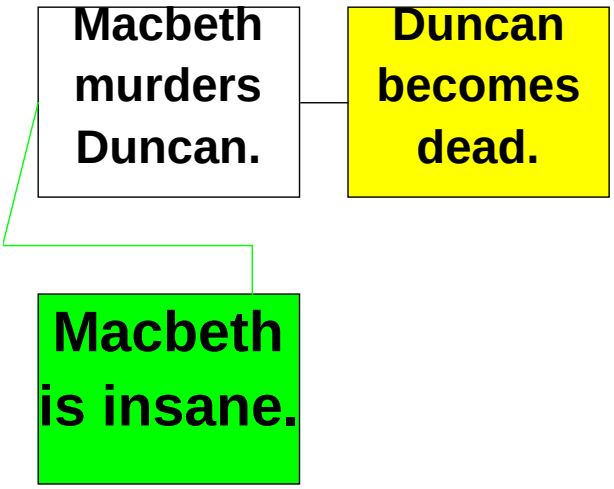

Figure 6: An abduction rule assumes the apparent consequent is actually an antecedent.

```
xx, yy, and zz are persons.
// A prediction rule:
If xx kills yy then yy becomes dead.
// An explanation rule; note use of "may":
If xx angers yy, then yy may kill xx.
// A presumption rule; note use of "can be"
If xx is foolish, xx can be greedy.
// An abduction rule; note use of "must be"
If xx kills yy, xx must be insane.
// An enablement rule; note use of "enables"
Xx's having a knife enables xx's stabbing yy.
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
Macbeth stabs Duncan.
The end.
```

The result is shown in the elaboration graph shown in figure 7:

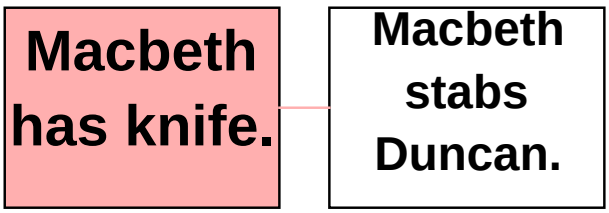

Figure 7: An enablement rule asserts that a required condition must hold because a certain type of action occurred.

Post hoc ergo propter hoc rules

When two events are proximate, we can indicate a causal connection using a post hoc ergo propter hoc rule, as in:

```
Start experiment.
```

```
xx, yy, and zz are persons.
// A prediction rule:
If xx kills yy then yy becomes dead.
// An explanation rule; note use of "may":
If xx angers yy, then yy may kill xx.
// A presumption rule; note use of "can be"
If xx is foolish, xx can be greedy.
// An abduction rule; note use of "must be"
If xx kills yy, xx must be insane.
// An enablement rule; note use of "enables"
Xx's having a knife enables xx's stabbing yy.
// A post hoc ergo proper hoc rule; note use of semicolon
Xx becomes king; yy becomes angry.
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
Macbeth becomes king.
Macduff becomes angry.
The end.
```

Because *Macduff's becoming angry* directly follows *Macbeth becomes king*, the rule yields the result shown in figure refposthoc. Note that if there are any sentences in between, the rule does not apply.

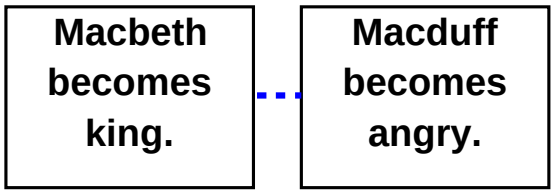

Figure 8: A post hoc ergo propter hoc rule, noting proximity, presumes cause.

Censor rules

A censor rule prevents inference and explanation rules from making ridiculous conclusions. For example, if you murder someone, you harm them. If you harm someone, that person becomes unhappy. But if you murder someone, that person cannot come to have some emotional state because that person is dead, even though an inference rule thinks that person should become unhappy.

The solution is to introduce a censor rule:

```
If YY becomes dead, then YY cannot become unhappy.
```

Such a rule, indicated by *cannot*, prevents any other rule from asserting that someone is unhappy after that person becomes dead.

```
Start experiment.
xx, yy, and zz are persons.
// A prediction rule:
If xx kills yy then yy becomes dead.
// An explanation rule; note use of "may":
If xx angers yy, then yy may kill xx.
// A presumption rule; note use of "can be"
If xx is foolish, xx can be greedy.
// An abduction rule; note use of "must be"
If xx kills yy, xx must be insane.
```

```
// An enablement rule; note use of "enables"
Xx's having a knife enables xx's stabbing yy.
// A post hoc ergo proper hoc rule; note use of semicolon
Xx becomes king; yy becomes angry.
// A censor rule; note use of cannot
If xx becomes dead, then xx cannot become unhappy.
// Prediction rules that would otherwise make a person unhappy if killed.
If xx kills yy then xx harms yy.
If xx harms yy then yy becomes unhappy.
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
Macbeth murders Duncan.
The end.
```

Not same as statements

Sometimes, as in the case of revenge, you want to insist that variables have different bindings as follows:

```
Start description of "Revenge".
xx is an entity.
yy is a entity.
xx's harming yy leads to yy's harming xx.
xx must not equal yy.
The end.
```

Without the *xx must not equal yy*, suicide could be an act of revenge because, convolutedly, Lady Macbeth's suicide harms herself, which harms Macbeth, a relative, which harms Lady Macbeth, a relative, so harm leads to harm.

Explicit connections

Cause expressions

Of course, stories can contain explicit causal connections. This one, for example:

```
Start experiment.
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
Macbeth wants to murder Duncan because Macbeth wants to become king.
The end.
```

The result appears in the elaboration graph, which now has an explicit connection, as shown in figure 9.

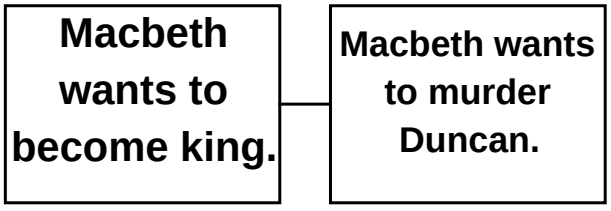

Figure 9: Explicit cause expressions connect antecedents to consequents.

Leads to expressions

You can also use *leads to* expressions in stories, meaning there is a causal connection, but the details are not given, as in:

```
Start experiment.
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
Macbeth's murdering Duncan leads to Macduff's fleeing to England.
The end.
```

The result appears in the elaboration graph, which now has an leads-to connection, as shown in figure 10.

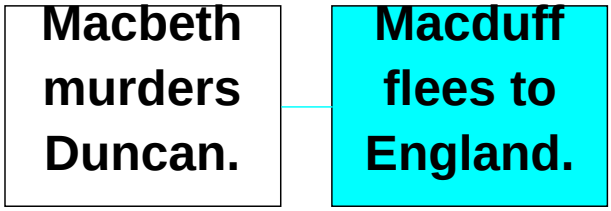

Figure 10: Leads to means there is a causal connection, but the details are not known.

Strangely leads to expressions

If you wish, you can add *Strangely*, indicating that the details are unknowable, which comes up, for example, in Victor Yarlott's work on Crow folklore:

```
Start experiment.
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
Strangely, Macbeth's murdering Duncan leads to Macbeth's hallucinating.
The end.
```

The result is shown in figure 11.

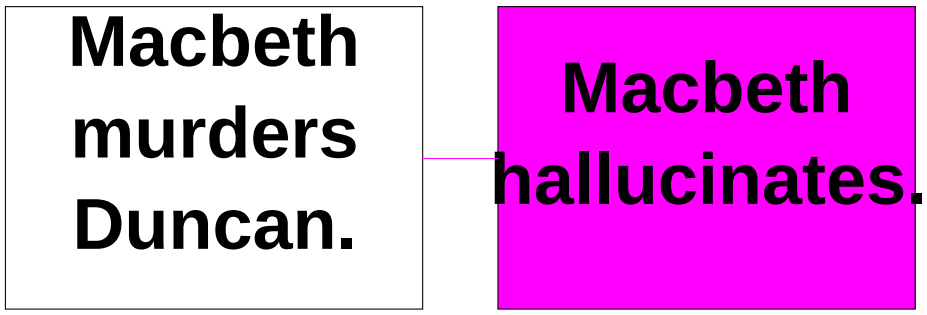

Figure 11: Unknowable leads to means there is a causal connection, but the details are not knowable.

In order to expressions

Genesis treats means as a kind of causal connection because the means enables the action, as in the following:

```
Start experiment.
Start story titled "Tragedy of Macbeth".
Macbeth, Macduff, and Duncan are persons.
In order to murder Duncan, Macbeth kills the guards and stabs Duncan.
The end.
```

The result is shown in figure 12.

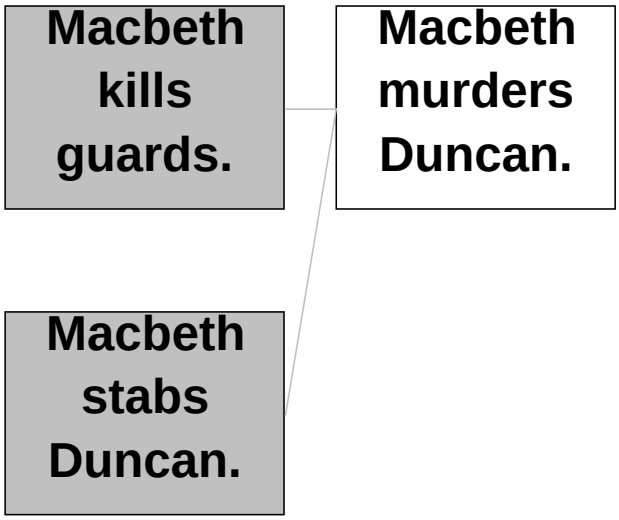

Figure 12: Means expressions show how actions are accomplished.

Probabilities

Ordinary entities as well as prediction rules and explanation rules can have probabilities attached. The attachment is expressed as in the following examples (at this writing, if forms do not parse).

With probability of 0.2 xx becomes dead.

With probability of 0.7 xx becomes dead because yy stabs xx.

With probability of 0.2 xx may kill yy because yy angers xx.

What you do with these probabilities is for you to decide, as at the moment, nothing uses them. Note that the entity getter, `getProbability`, fetches an entity's probability if there is one, or `null` otherwise.

Concept patterns

Basic concept patterns feature leads-to relations

Concept patterns usually include specifications for searches expressed as *leads to* expressions:

```
Start experiment.
xx and yy are persons.
Start description of "Revenge".
xx's harming yy leads to yy's harming xx.
The end.
Lady Macbeth, Lady Macduff, Macbeth, Macduff, and Duncan are persons.
Start story titled "Macbeth".
// Of course, most of the following connections would be placed
```

```
// by prediction and explanation rules.
Macbeth harms Duncan because Macbeth Murders Duncan.
Macbeth harms Macduff because Macbeth harms Duncan.
Macbeth angers Macduff because Macbeth harms Macduff.
Macduff wants to kill Macbeth because Macbeth angers Macduff.
Macduff kills Macbeth because Macduff wants to kill Macbeth.
Macduff harms Macbeth because Macduff kills Macbeth.
The end.
```

Note that a concept pattern looks like a story, except that the opening idiom, *Start description of ...* . is different from *Start story titled ...* .

The *leads to* part of the concept pattern says that the first event must be causally connected to the second event but there may be any number of causal links in between. For example, there are three causal links connecting *Macbeth harms Macduff* to *Macduff harms Macbeth*, as shown in figure 13.

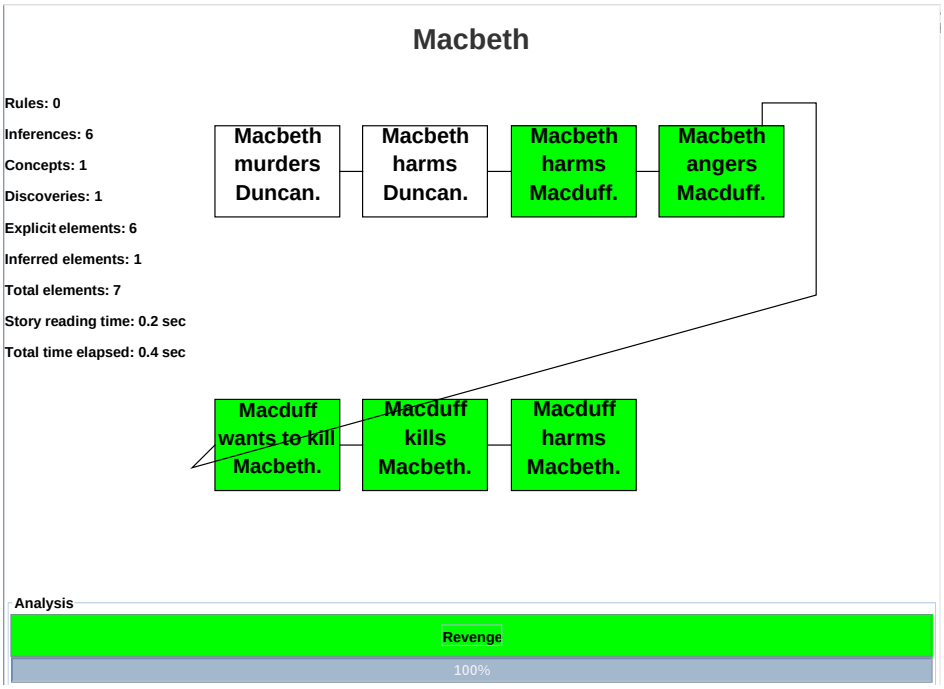

Figure 13: An instantiated concept pattern. Note that any number of causal connections may be involved.

You see that the *Analysis* line in the user interface indicates that Genesis has identified the *Revenge* pattern as a concept, even though the word *murder* did not appear in the story.

Note that Genesis is rigid about the way the leads to relations are expressed:

```
yy's angering xx leads to xx's murdering yy.
```

works, but

```
yy angering xx leads to xx murdering yy.
```

does not work. The difference is a consequence of limitations in the START parser.

Generalizations

Inference rules, explanation rules, and censor rules all may have multiple antecedents, as in this inference rule:

```
If XX is king and WW is XX's successor and XX becomes dead,
then WW becomes king.
```

Inference rules may not have multiple consequents, however. You would need to express such a rule as multiple rules, one for each consequent.

Concept patterns may have multiple *leads-to* expressions:

```
Start description of "Pyrrhic victory".
xx is an entity.
ll is an action.
zz is an entity.
xx's wanting ll leads to xx's becoming happy.
xx's wanting ll leads to zz's harming xx.
The end.
```

This particular example also illustrates the incorporation of an unspecified action, *ll*.

Concept patterns may also have ordinary relations and events that are not involved in *leads-to* relations. In the following, *yy is an enemy of zz.* and *I am a friend of yy.* are relations and *yy helps xx.* is an event:

```
Start description of "Sell out".
xx is an entity.
yy is a entity.
zz is a entity.
yy is an enemy of zz.
yy helps xx.
I am a friend of yy.
zz's helping xx leads to xx's angering yy.
The end.
```

Concept patterns may also have *sometimes* elements, meaning that the element is optional. If the element is present, it is included in the concept discovered, but it need not be present. Another possibility is the specification of conclusions to be reached, and put back in the story, as a result of a matched concept:

```
Start experiment.
xx and yy are persons.
Start description of "Revenge".
xx's harming yy leads to yy's harming xx.
// An optional element; note use of "sometimes"
Sometimes xx hates yy.
// A conclusion; note use of "Consequently"
Consequently, yy becomes exonerated.
The end.
Lady Macbeth, Lady Macduff, Macbeth, Macduff, and Duncan are persons.
Start story titled "Macbeth".
// Of course, most of the following connections would be placed
// by prediction and explanation rules.
Macbeth harms Duncan because Macbeth Murders Duncan.
Macbeth harms Macduff because Macbeth harms Duncan.
Macbeth angers Macduff because Macbeth harms Macduff.
Macduff wants to kill Macbeth because Macbeth angers Macduff.
Macduff kills Macbeth because Macduff wants to kill Macbeth.
Macduff harms Macbeth because Macduff kills Macbeth.
The end.
```

The result is shown in figure 14.

.

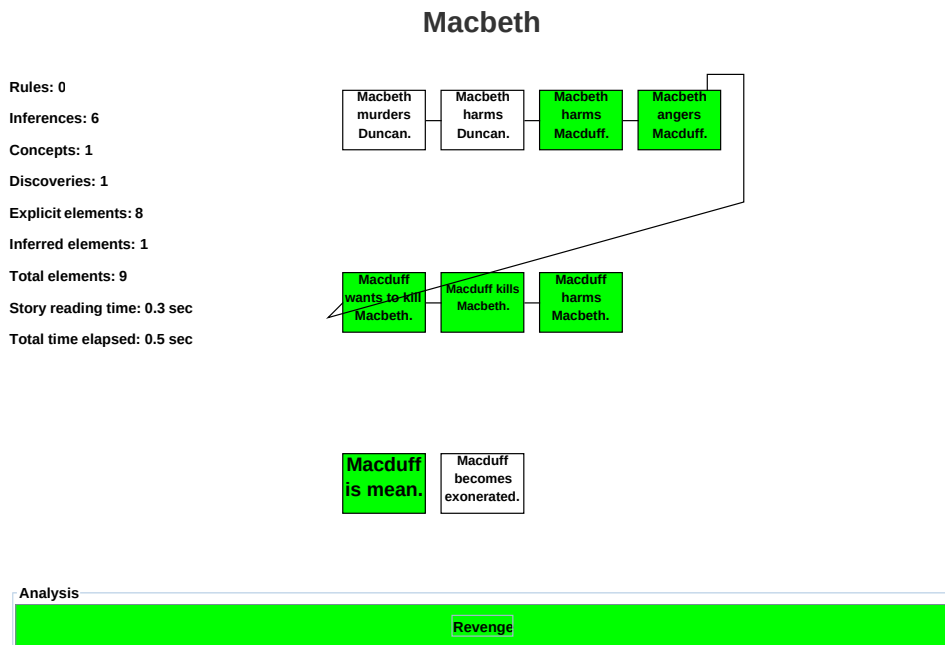

Figure 14: An instantiated concept pattern with an optional element, *Macduff is mean*, and a consequence inserted back into the story, *Macduff becomes exoneration*

Another example came up in handling Crow folktales, as `xx has strong medicine` is instantiated from pattern variables previously bound and inserted back into the story if the ”Violated belief - Medicine Man” concept is discovered.

```
Start description of "Violated belief – Medicine Man".
xx is a person.
yy is a thing.
xx transforms yy.
Consequently, xx has strong medicine.
The end.
```

Finally, you can insist that two variables have different values using a *must not equal* expression. For example, you can insist that nothing can be involved in revenge against itself:

```
Start description of "Revenge".
xx is an entity.
yy is a entity.
xx's harming yy leads to yy's harming xx.
xx must not equal yy.
The end.
```

When *is* becomes *becomes*

Early in the development of Genesis, a conundrum appeared. If you say *Macbeth is happy.*, does that mean Macbeth has been happy since the beginning of time or that he has just become happy? We humans sort it out with a level of common sense not yet in Genesis.

The solution is to say *become* explicitly when a property just appears.

Scenes

Ordinarily, if you write `Boris insults Patrick` in a story twice, it only shows up once because it is assumed to be the same insult. You can have it appear twice, however, by starting a sentence with "Then, ...", which puts in a scene marker and allows the same innerese expression to appear more than once, as in this story, with the result shown in figure 15

```
Start story titled "Duplicates".
Boris insults Patrick.
Boris insults Patrick.
Then, Boris insults Patrick.
The end.
```

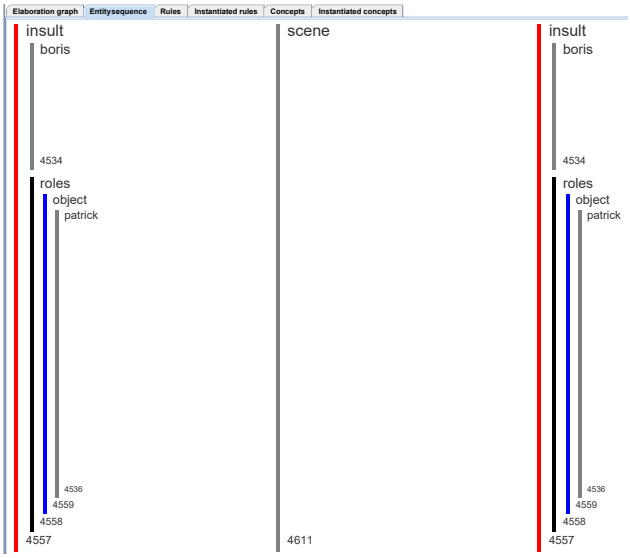

Figure 15: Using `Then, ...` constructions inserts a scene marker and allows repetition. The second insult is not recorded in the story, but the third one is, because it is in a new scene.

File reader idiom

Naturally, you will want to assemble experiments using multiple files, perhaps one for common sense rules, another for concept patterns, yet another for the story. This is easy to arrange using the file reader idiom, as in the following examples. Note that *.txt* is assumed by the idiom processor:

```
// Find and read file named "General commonsense knowledge.txt"
Insert file General commonsense knowledge.
// Find and read file named "General commonsense knowledge.txt"
Insert file General reflective knowledge.
```

But where are those files? Genesis does you a favor and looks around locally in your file structure to find them. If there are multiple files with the same name, you need to be careful, of course. Genesis will warn you that there are such name conflicts on the console, and pick one file from the alternatives, but you won't know if you ignore the console.

Story processor idioms

Each experiment should start with the following idiom, which clears text boxes and memories from previous experiment:

```
Start experiment
```

Another general purpose idiom primes the text entry box. For example, the following primes the text entry box with *Make Macbeth be nice*:

```
Insert into text box: Make Macbeth be nice.
```

You can put this anywhere in a file.

Perspective idioms

Genesis has two primary story processors, and various idioms tell Genesis where to direct the text and what to show on the user interface.

```
Both perspectives.  // Direct text to both story processors.
First perspective.  // Direct text to left-side story processor only.
Second perspective. // Direct text to right-side story processor only.


Show both perspectives.  // Show both perspectives.
Show first perspective.  // Show left-side elaboration graph only.
Show second perspective. // Show right-side elaboration graph only.
```

Switch idioms

Genesis has lots of switches, many of which can be set by idiom. At this writing, all the switches in the Subsystems panel, the Story summarizing panel, the Story persuasion panel, the Why questions panel, and the Presentation panel can be set by idiom, as suggested by the following examples.

```
Set Show markup switch to true.
Set Summarizer switch to false.
```

User interface idioms.

The genesis user interface has three panes. Which information goes to which pane is under idiom control. For example:

```
Set left panel to controls.
Set right panel to mental models.
Set bottom panel to elaboration graph.
```

Thus, the patterns are *Set left panel to ...*, *Set right panel to ...*, and *Set bottom panel to ...* where the ellipses are replaced by the name of a item that can be displayed. All the possibilities are exposed when you click on the three vertical bars on the top left of each pane, as shown in figure 16:

Figure 16: Clicking on the three bars allows you to pick the panel to be displayed.

Ordinary classification

If you want to associate a name with a single thread, write as follows:

```
John is a person.
```

The result is that John has just one thread:

```
thing entity physical-entity object whole living-thing organism person name john
```

Note that the final element in the thread is the name of the entity; the penultimate element is a marker that signals a name. This convention is exploited by the matcher.

On the other hand, if you want a name to have an additional thread, you write:

```
John is also a criminal.
```

The result is that John has an added thread:

```
thing entity physical-entity object whole living-thing organism person name john
thing entity physical-entity object whole living-thing organism person bad-person
wrongdoer principal criminal name john
```

Again, note that the final elements in the threads are the name of the entity; the penultimate element is a marker that signals a name.

Finally, maybe you want to define a class not in wordnet:

```
A bouvier is a kind of dog.
```

The result is:

```
thing entity physical-entity object whole living-thing organism animal chordate
vertebrate mammal placental carnivore canine dog bouvier
```

Note that there is no name marker.

New names

Suppose you want to introduce a name, such as SuspectA, that START does not know about. Here is what you could do:

```
Note that SuspectA is a name.
SuspectA is a suspect.
```

Generally, however, it is better to supply a gender. Otherwise the use of pronouns will not work, as in "SuspectA killed himself;" the problem is that if you do not supply gender, SuspectA will be neuter and START will not substitute SuspectA for himself. So, here is how you supply gender:

```
Note that SuspectA is a masculine name.
```

or

```
Note that SuspectA is a feminine name.
```

Classification idioms

Sometimes you will want to specify a complete thread for a word or phrase. This first come up in persuasive story telling when Sila Sayan wanted to declare many properties that make someone likable or unlikable.

```
Assert thread thing, likable, good parent.
Assert thread thing, likable, caring.
```

It also became useful to be able to specify opposites:

```
Likable is the opposite of unlikable.
```

Specifying threads.

Suppose you say:

```
A rat scared Mary.
```

Alas, wordnet supplies five meanings for rat, three of which are subclasses of people. To order Genesis to limit itself to the three, you can use the following convention:

```
A (rat person) scared Mary.
```

And if you really want to be specific, you can write:

```
A (rat unpleasant-person) scared Mary.
```

Quotation

Sometimes START needs a little help when sentences get deeply nested. You can do this using nested s̈ and <> pairs as in the following example:

```
John believes "Mary thinks <Sally knows "Susan loves Paul">".
```

We use the brackets instead of matched single quotes because the use of single quotes for possession would confuse Genesis terribly. The result of the demonstration sentences is as in figure 17.

Geneses also provides a bit of syntactic sugar.

```
John: Mary is crazy.
```

is equivalent to

```
John says "Mary is crazy".
```

Deprecated idioms

You may see in files various idioms that were once useful, but now are ignored or replaced, as this one, for example:

```
Insert file Start experiment
```

The story processors used to be initialized by reading idioms from a file. Now, this idiom merely does the same thing as `Start experiment.` which involves no file reading.

```
Start commonsense knowledge.
```

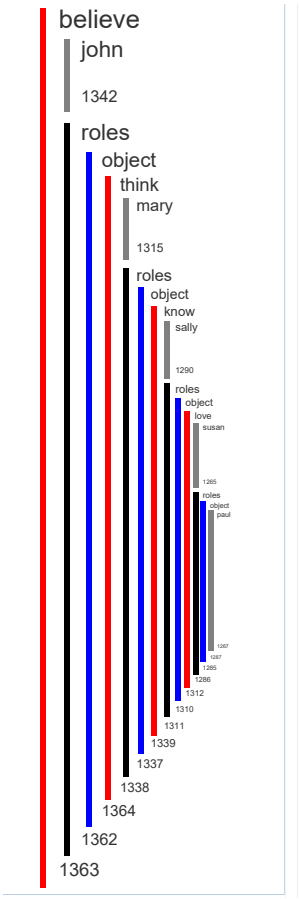

Figure 17: Nested quotations enable elaborate constructions.

This idiom used to tell the user interface that the next lines of text were part of the common sense collection. This is now automatic and the idiom is ignored.

```
Start reflective knowledge.
```

This idiom used to tell the user interface that the next lines of text were part of the concept pattern collection. This is now automatic and the idiom is ignored.