APPENDIX A

THE GRAIL THEOREM PROVER

THIS appendix gives an overview of the Grail system, developed as part of my PhD project, and its use as a tool for the development and prototyping of grammar fragments for the multimodal Lambek calculus.

Grail is an automated theorem prover based on proof nets and algebraic labeling, a combination discussed in Chapter 6. The theorem prover is implemented in SICStus Prolog, the user interface in TclTk.

Though the underlying logic, with a minor restriction on the structural rules, is decidable, and the theorem prover can operate automatically, user guidance is often desirable during the proof search. It can increase the performance of the algorithm and, more importantly, help the user visualize the status of the proof attempt thereby showing *why* a given statement is provable or not.

The Grail user interface is based on the Prolog debugger. At each proof step the user can take one of the following actions: *select* allows the user to select an inference step, *leap* performs automatic proof search until a proof is found, *fail* marks the current branch of the search tree as unsuccessful and *abort* abandons the entire proof attempt.

In my experience, the interface gives users better insight in the operation of the theorem prover and greatly enhances its facilities for prototyping and debugging of fragments of the multimodal Lambek calculus.

A.1 History

In the end of 1995, the first incarnation of Grail was a piece of Prolog code of some 250 lines. You could enter a logical statement and wait until it produced an answer in the form of `yes` or `no`, or until you got bored (which happened

a lot those days). In spite of a number of improvements to the efficiency of the original code, several grammar fragments designed in it could not handle longer sentences in a reasonable amount of time.

I therefore tried to give a user-friendly representation of the computation state with which the user can inspect and guide the computation. The benefits of this are twofold: firstly, the user can select a promising state from the possible states and abandon hopeless subgoals, and secondly, it gives the user insight into *why* specific statements are underivable, without having to use the Prolog debugger where tracing the execution is difficult even for the programmer.

The current version is some 8000 lines of mixed Prolog and TclTk code, using the TclTk library included with SICStus Prolog. It can be used without knowledge of Prolog and produces output in human-friendly natural deduction format.

Grail is used as a research tool and as courseware for introductory to advanced level courses in Lambek grammars. Grail is free software distributed under the GNU General Public License, and can be downloaded as source code and binaries from my personal home page.

```
http://www.let.uu.nl/~Richard.Moot/personal/grail.html
```

A.2 Tutorial

Before I give an in-depth overview of all possibilities at the different windows in Grail, it is perhaps useful to give a short introduction to designing and running grammar fragments in Grail.

A.2.1 Getting Started

You start Grail from the directory where you installed the source code or the binaries by giving the follow command.

```
sicstus -l grail
```

This will start Grail and, if SICStus and TclTk are correctly installed on your system, the window shown in Figure A.1 will appear.

This is the main window, where you activate the theorem prover and open new windows to edit the current grammar fragment. See Section A.3.1 for an overview of all options here.

A.2.2 The Lexicon

Since we start with an empty grammar fragment, we will first fill the lexicon with a few useful words. From the main window, select [Window/Lexicon Window] to open the lexicon window. The lexicon, being empty, does not

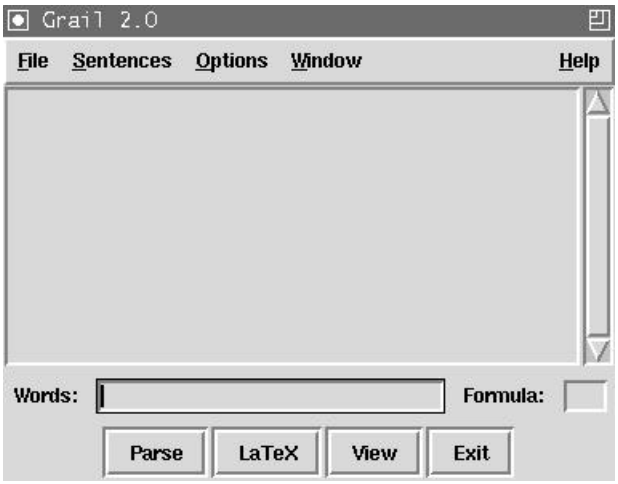

Figure A.1: The Grail startup window.

look very interesting right now, so select [Edit/New Entry...] from the meny bar of the lexicon window. The following window should appear.

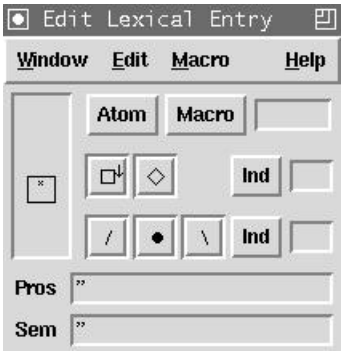

Figure A.2: The lexicon edit window.

Lexical Entries A lexical entry in Grail consists of three things: a prosodic entry, representing the word being described, a formula and a semantic entry, which encode the lamba term meaning of the current entry. The entire top part of the lexicon edit window is dedicated to entering the formula.

Simple Formulas We start by assigning a simple np formula to a word in the lexicon. Because no atomic formulas have been defined for our fragment yet, we type `np`, followed by <Enter> in the text entry field next to the

[Atom] and [Macro] selection fields. You can click on [Atom] to verify that np has been added as an atomic formula. Enter 'tony' in both the Pros and the Sem entry field. The lexical edit window should now look as shown in Figure A.3.

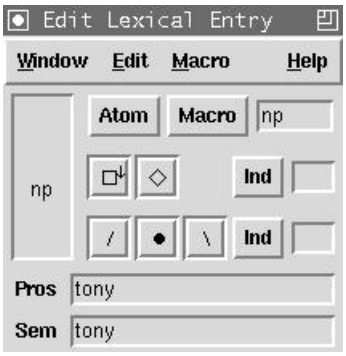

Figure A.3: The lexicon edit window after typing in the first lexical entry.

Storing Entries It is important to note that the edits in the lexical edit window functions will only affect the lexicon once you select [Edit/Store Entry] from the menu. If you select [Edit/Store Entry] from the menu now, you will see the entry for 'tony' appear in the lexicon window.

Exercise 1 *Add some more np's to the lexicon by editing the Pros and Sem fields followed by [Edit/Store Entry] until the lexicon looks as in Figure A.4.*

Complex Formulas Now, we will add some entries with complex formulas to the lexicon, assigning the formula $(np\backslash_a s)/_a np$ to 'shot'. First, erase the previous lexical entry by selecting [Edit/Clear Entry] from the menu, then type 'shot' in both the Pros and Sem fields and 'a' in the index field next to the binary connectivies as shown in Figure A.5.

Formulas are entered top-down left-to-right, always starting with the main connective of the current (sub)formula, and always specifying the left subformula before the right subformula. For $(np\backslash_a s)/_a np$ the main connective is '$/_a$', so we proceed by pressing the [/] button. The result is shown in Figure A.6.

We continue with the left subformula, which is the complex formula $np\backslash_a s$ with main connective '$\backslash_a$'. The correct index a should still be in the index field, if not, reenter it or select it from the [Index] menu next to it. Pressing the [\] button now should result in Figure A.7.

We've entered all connectives now and we only have to fill in the atomic formulas. We can do this be entering them in the atom field and pressing <Enter> or by selecting them from the [Atom] menu if we've entered them

Figure A.4: The lexicon after inputting some np's.

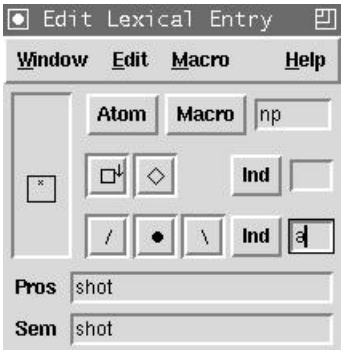

Figure A.5: The edit window after entering the mode information.

before. After selecting 'np', entering 's' and selecting 'np' again, our formula looks as shown in Figure A.8 and we can select [Edit/Store Entry] to store it in the lexicon.

Exercise 2 *Add entries for 'likes', 'hates' and 'distrusts' to the lexicon.*

The Macro Facilities You can mark any subformula of the current lexical entry by clicking it and give it a name to use later. For example, we can click on the main connective '$/_a$' of the lexical entry for 'shot' to select the entire '$(np\backslash_a)/_a np$' formula, we can give it the abbreviation 'tv', for transitive verb, by entering this in the atom field an selecting [Macro/Store Selection

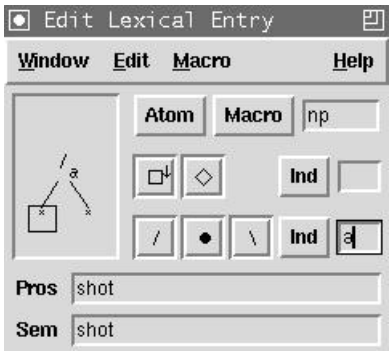

Figure A.6: The edit window after entering '$/_a$'.

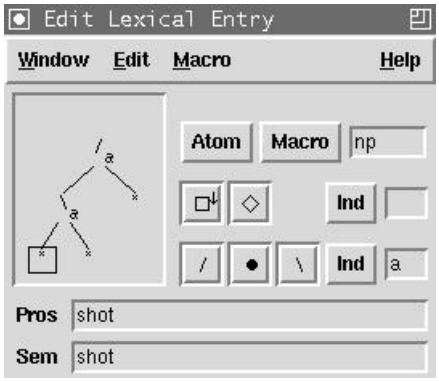

Figure A.7: The edit window after entering '$\backslash_a$'.

as Macro] from the menu. We can also click on the connective '$\backslash_a$' to select the formula $np\backslash_a a$ and store it as 'iv' for intransitive verb, again by entering this in the atom field and selecting [Macro/Store Selection as Macro].

We can now enter more complex entries quite simply. Suppose we want to assign the word 'himself' the formula '$((np\backslash_a s)/_a np)\backslash_a (np\backslash_a s)$'. With the macro's we just stored this is equivalent to '$tv\backslash_a iv$', so we can press [\], select 'tv' from the [Macro] menu, then select 'iv' from the macro menu and we have entered the correct formula. After storing it, the lexicon should look as shown in Figure A.9.

The macro's are also the simplest way of producing complex goal formulas for use in the main window.

Exercise 3 *Give 'someone' an entry of the form $s/_a iv$. Store this entry as a macro for 'gq_s', a subject generalized quantifier.*

Exercise 4 *Use the macro from the previous exercise to assign a lexical entries of*

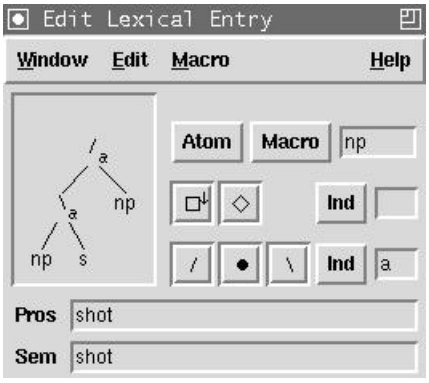

Figure A.8: The edit window after entering all atomic formulas.

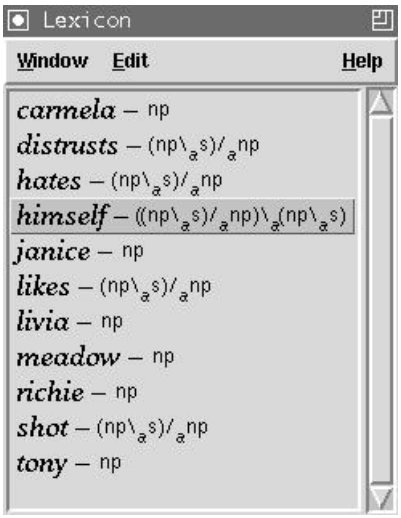

Figure A.9: The lexicon window after entering some complex formulas.

the form 'gq_s/$_a$n' to 'a' and 'every'.

Correcting Mistakes When you notice you have made a mistake when entering the current formula there are several ways of correcting it.

First of all, if you want to erase the lexical entry window completely, you can select [Edit/Clear Entry] from the menu.

If you want to erase the currect formula, or a part of it, while keeping the [Pros] and [Sem] fields intact, you can select the root of the formula tree you want to erase and select [Edit/Cut] from the menu or press (Control-k) on the keyboard. This will replace the currently selected formula tree by

an insertion point. Selecting [Edit/Paste] or pressing (Control-y) while you have selected an insertion point will paste the formula you have cut at the insertion point. Selecting [Edit/Copy] or pressing (Control-c) will store the selected formula for the next paste operation without deleting anything.

Finally you may want to replace a connective with a different connective, this is done simple by selecting the connective and pressing one of the connective buttons. Replacing '/' by '\' or vice versa will also switch the order of the subformulas, all other replacements will leave the order of the subformulas unchanged.

It is not possible to replace an atom with a different atom or a connective in this way: you have to erase this explicitly with [Edit/Cut] or (Control-k).

If you notice that you have stored an incorrect entry into the lexicon and you wish to correct it, the best way to proceed is to double-click on the lexical entry or to click on it followed by selecting [Edit/Edit Entry...] from the lexicon menu. This will open the incorrect entry into the lexical edit window. Then, select [Edit/Delete Entry] from the lexicon menu or press (Control-Button 1) to delete the incorrect entry from the lexicon, and proceed by editing the incorrect entry in the lexical edit window.

Exercise 5 *Use the edit facilities to change a transitive verb from the lexicon to produce an entry for 'talks' of the form '$(np\backslash_a s)/_a pp$' and an entry for 'needs' of the form '$(np\backslash_a s)/_a((s/_a np)\backslash_a s)$'.*

Exercise 6 *Edit the lexical assignment to 'someone' from Exercise 3 to produce an additional lexical entry of the form '$(s/_a np)\backslash_a s$'. Store this formula as a macro for gq_o, an object generalized quantifier. Use this macro to assign appropriate new lexical entries to 'a' and 'every' as well.*

Loading and Saving Your Fragment Now that we have a simple lexicon, it is time to put the theorem prover to work, but before we do that, we first save the grammar fragment we have produced so far to a file. Select [File/Save Fragment...] from the menu of the main window, then type in a file name in the entry field and press the [Save] button.

To load this fragment again from a new Grail session select [File/Consult Fragment...] from the menu, select the file from the file select window, then press the [Open] button.

Saving or loading a file will change the name of the main Grail window to the name of the current file.

A.2.3 The Theorem Prover

Entering a New Sentence From the main window, type a sentence in the entry field marked 'words' and a formula in the entry marked 'formula'. The sentence can be any string.

Lexical Lookup By default, Grail does not distinguish between upper and lower case and all interpunction symbols are treated as spaces. The reference guide details how to modify this standard behavior. Press <Enter>, select [Sentences/Parse] from the main window menu or press the [Parse] button to start the theorem prover. The status window will now appear.

Any string between spaces is treated as a word and will be looked up in the lexicon. If a word has no lexical entries at all, Grail will complain and abort the attempt. Check the lexicon and your spelling if this happens.

Proof Status If lexical lookup succeeds, the status window will appear to give updates on the status of the proof attempt and show a rough estimate of the computation remaining for the current lookup.

While the theorem prover is running, it is not possible to edit the lexicon or any other aspect of the current grammar. However, if you get tired of waiting, you can press the [Abort] button from the status window or use (Control-c).

After the theorem prover has completed its computations, the status window will have one of the following messages.

[Done] One or more solutions were found.

[Failed] No solutions were found.

[Aborted] User aborted the computation.

The sentence will now be added to the list of sentences on the main window. If no solutions were found before the user aborted the computation, the sentence will have a '?' prefixed to it. If no solutions we found even though the computation finished, the sentence will have a '*' prefixed to it. Otherwise, the sentence will have a space as its first symbol. Keep in mind that Grail only looks remembers the proof attempt for a sentence: retrying after changing the grammar fragment can result in new derivability markings.

You can retry a sentence without entering it again by double-clicking on it, or by selecting it and the pressing [Parse], the <Enter> button or selecting [Edit/Parse] from the menu.

Exercise 7 *Enter the sentences shown in Figure A.10 in the main window. The goal formula in all these cases is* s.

Proofs If you have LaTeX installed on your computer, Grail can produce natural deduction output for any proofs it has found. Press the [View] button will cause you selected LaTeX previewer to appear with one or more natural deduction proofs for the last successfully parsed sentence. For the different types of natural deduction output format, we will refer to the reference manual. A typical natural deduction proof produced by Grail is shown in Figure A.11.

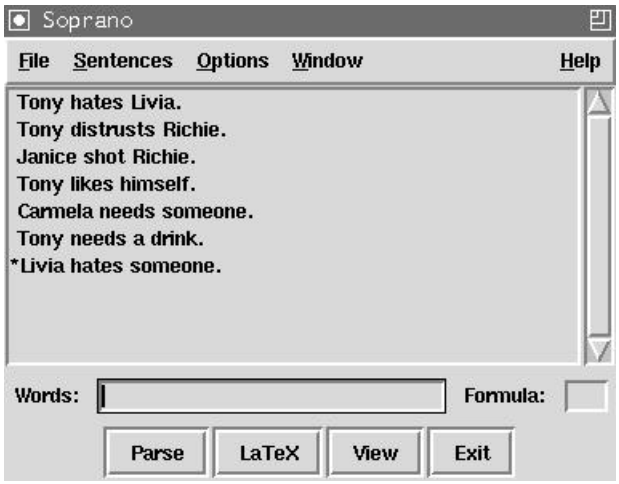

Figure A.10: The lexicon window after entering some complex formulas.

$$\dfrac{\text{tony} \vdash np \qquad \dfrac{\text{distrusts} \vdash (np\backslash_a s)/_a np \quad \text{richie} \vdash np}{\text{distrusts} \circ_a \text{richie} \vdash np\backslash_a s}\,[/E]}{\text{tony} \circ_a (\text{distrusts} \circ_a \text{richie}) \vdash s}\,[\backslash E]$$

1. ((**distrusts richie**) **tony**)

Figure A.11: LaTeX natural deduction output

Debug Mode There can be cases where you want to get more detail as to why a certain sentence was underivable in the current fragment, or — equally important — to guide the theorem prover to a proof which would take too long to find without guidance. For these situations, Grail has a debug mode, which you can turn on by selecting [Debug/Interactive] from the status window. This will cause the status window to expand to the proof net window.

Double-click on the sentence 'Livia hates someone'. Grail has marked it as underivable and we want to know why. The proof net window should look like shown in Figure A.12.

The proof net window displays the formulas in a way similar to the lexical edit window, with the following differences.

- the main connective of a formula is at the bottom.
- positive atomic formulas are drawn in white, negative atomic formulas are drawn in black.
- par links are drawn in dotted lines.

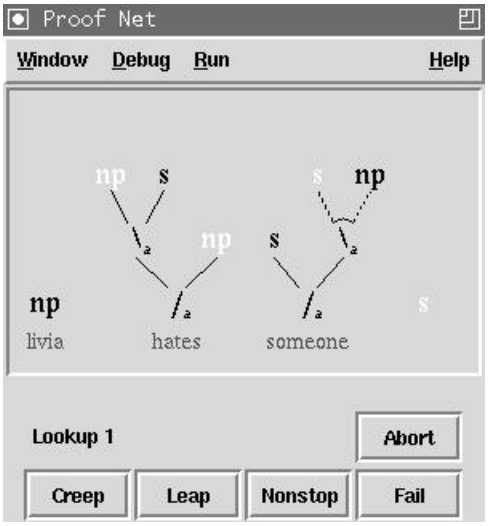

Figure A.12: The proof net window for 'Livia hates someone'.

The links used in the proof net window are essentially the same as those we used for the Lambek calculus in Section 4.7. This makes it easy to identify proof nets which need some form of the commutativity rule, as those proof nets will have crossing axiom links.

The lookup shown in Figure A.12 uses the subject generalized quantifier type for 'someone', which seems unlikely to be correct. We can press the [Fail] button to reject this lookup and force Grail to find new formula assignments to the words of the sentence. Using [Fail] carelessly can lead to Grail failing to find proofs which it normally would have found; selecting [Fail] means *you* take responsability for the absence of proofs in the current branch of the search space. After pressing [Fail], Grail returns with the second lexical lookup and the proof net window looks as shown in Figure A.13.

The second lookup uses the object generalized quantifier type for 'someone', which appears the right choice in the current situation.

We can now select [Nonstop], which causes Grail to continue until it has found all proofs for the current sentence. However, because we already know the current proof attempt will fail, this is not the right choice.

We can also select [Leap], which causes Grail to continue until either.

- it has found a complete linking of all axioms.
- or it failed to produce a complete linking of all axioms at which point Grail will try to find new lexical assignments and, if successful, wait for your input again.

Selecting [Creep] takes us through the axiom links one at a time, though,

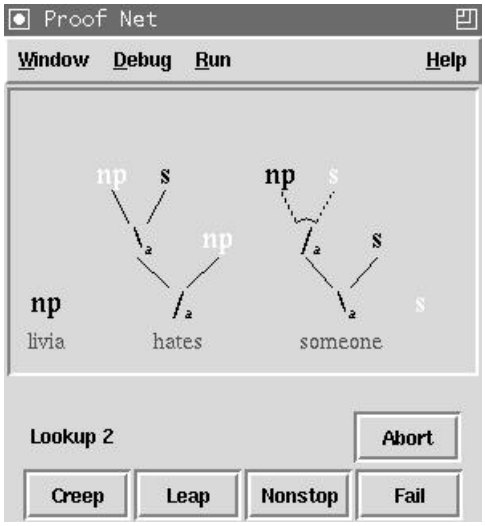

Figure A.13: The proof net window for 'Livia hates someone'.

by selecting [Fail] we can at any point mark the current linking as unsuccessful and continue at the next untried axiom link.

Manual Axiom Links The final possibility is to perform the axiom links manually. This is especially recommended in the case of larger proof nets. To make an axiom link manually, click one of the atomic formulas in the proof net window. Start with the leftmost, negative np which corresponds to 'Livia'. After clicking this np all possible positive np's you can link it to will be marked by a box, as shown in Figure A.14.

Select the leftmost positive np. An axiom link connecting the two np's will now appear. Grail will keep track of the other possibility for this axiom link for you, so you don't have to worry about mistakes. If at any point you produce an incorrect proof structure, for example by creating a cycle, Grail will complain and ask you to retry the last choice you made where alternatives were available.

If you are very sure there is only one correct way of linking the current formula, you can use <Shift> in combination with the left mouse button. This will commit you to the current axiom link. It is equivalent to selecting all other possible axiom links first and immediately following them by [Fail]. Be careful that this may prevent proofs from being found.

Exercise 8 *Connect all axiomatic formulas until the proof structure looks as shown in Figure A.15.*

After all axiom links have been made, you are given a final opportunity

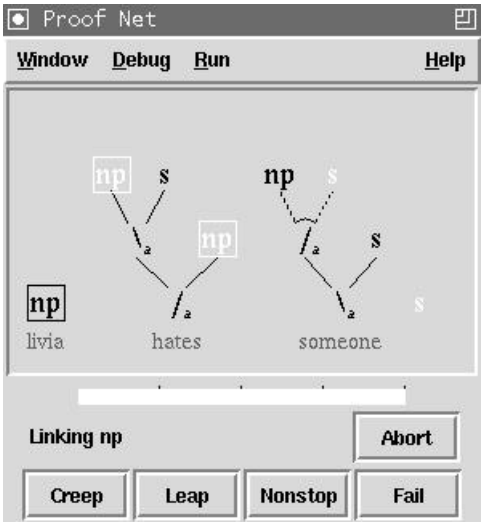

Figure A.14: Possibilities for linking the first *np*.

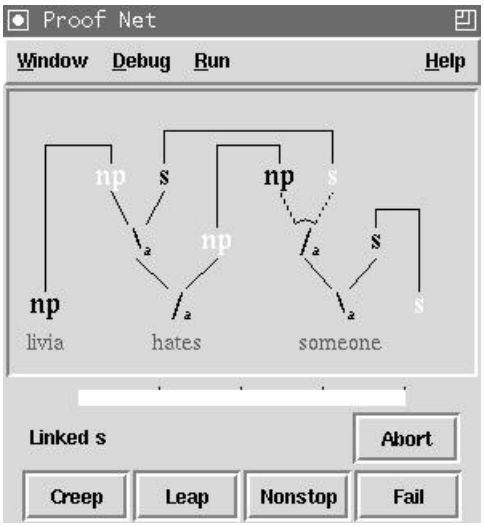

Figure A.15: Proof net after making all axiom links.

press [Fail] and try to find another linking. Pressing [Creep] or [Leap], however will take you to the rewrite window.

Rewriting The rewrite window contains the label computed for the current proof structure. Section 6.3 explains how the labels are obtained from acyclic

and connected proof structures. For the current proof structure, the label looks as shown in Figure A.16.

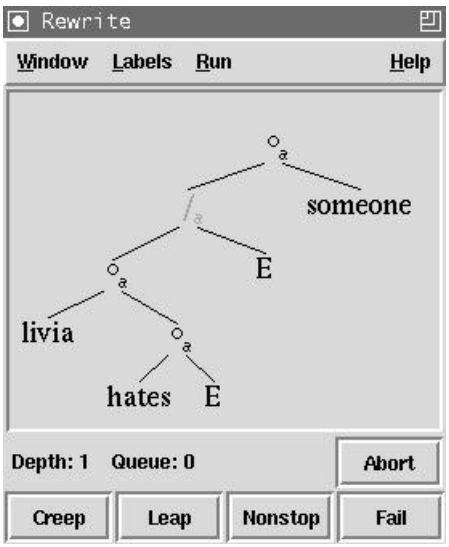

Figure A.16: Label for 'Livia hates someone'.

In order to produce a correct label, we have to do two things.

- remove all auxiliary constructors by means of their rewrite rules. For '/' we have to use to conversion shown in Figure A.17. The conversions for the other connectives as shown in Figure A.28.
- Left to right traversal to the label tree should produce the words in the order they appear in the input sentence.

In this case, the words are already in the correct order, but it is impossible at the moment to use the '/' conversion because we need to be in the configuration shown in Figure A.18 for that.

You can check for yourself that we cannot produce a correct label. [Leap] or [Creep] will not succeed and Grail will continue by trying to find a new

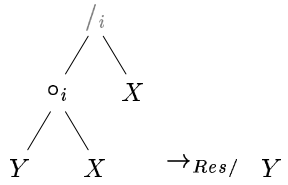

Figure A.17: Conversion for '/'.

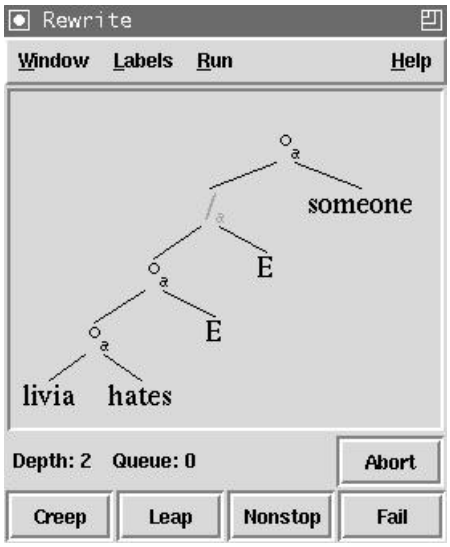

Figure A.18: Label we need for the '/' conversion.

axiom link. You can also click on every node of the tree to see a list pop up which shows which conversions are applicable at this node and see this list is empty for all nodes in the tree.

When you find a sentence which isn't derivable, while you would want it to be, you can

- either add a new entry to the lexicon for one of the words in the sentence or correct or modify an old entry
- or add a new structural rule to your grammar.

The first option seems unattractive in this case; we already have two assignments for the quantifier 'someone' and assigning a new formula for every new construction we encounter would lead to a huge lexicon. So in this case, we would like to generalize a bit by adding some structural rules.

A.2.4 The Structural Postulates

Editing a Postulate You open the postulate window from the main window by selecting [Window/Postulate Window] from the menu or by pressing (Control-p). Initially, your grammar fragment will have no structural postulates. But you can create a postulate by selecting [Edit/New Postulate] from the postulate window to make the postulate edit window appear, which looks as shown in Figure A.19.

The edit postulate window is similar in structure to the edit lexical entry window, only now we only have the '$\bullet$' and the '$\diamond$' as connectives and instead of atomic formulas we have structural variables.

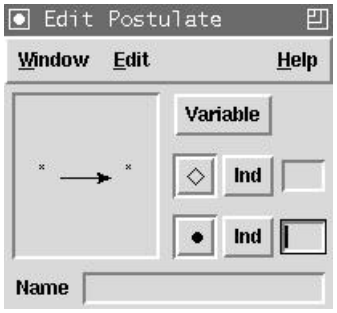

Figure A.19: Creating a new postulate.

To add a structural rule for associativity to the current grammar, first select a from the [Ind] menu next to the [•] button. Then click on the $*$ on the left hand side of the arrow and press the [•] button twice. The edit postulate window should now look as shown in Figure A.20.

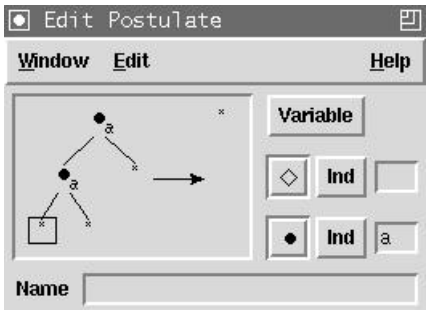

Figure A.20: Entering the left hand side of the postulate.

All that remains is to create the leaves. Select the [Variable] menu. Only one variable 'A' is available initially, so select it. Now, since 'A' has been used, a new variable 'B' will be added to the menu. Select this. Finally, select 'C' from the variable menu. We have now completed the left hand side of the structural rule.

Exercise 9 *Complete the right hand side of the postulate, so that it the postulate looks as shown in Figure A.21.*

Storing a Postulate Like the changes you make in the lexical edit menu, the changes in the postulate edit menu don't take effect until you select [Edit/Store Postulate] from the menu.

Before storing the postulate, let's give it a name by typing 'Ass' in the Name field. We have three possibilities for storing a postulate: we can store

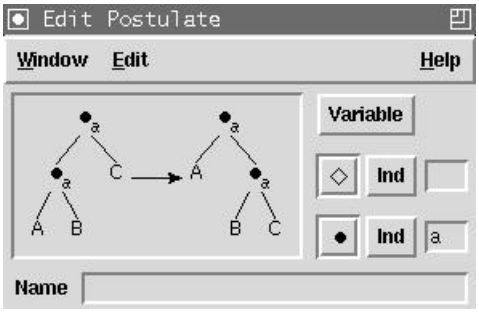

Figure A.21: Completed postulate.

the left-to-right version — which is the default —, we can store the right-to-left version or we can store both. We can toggle between these possibilities by clicking on the arrow. To save time, click on the arrow once then select [Edit/Store Postulate] store both versions of this postulate. The postulate window should now look as shown in Figure A.22.

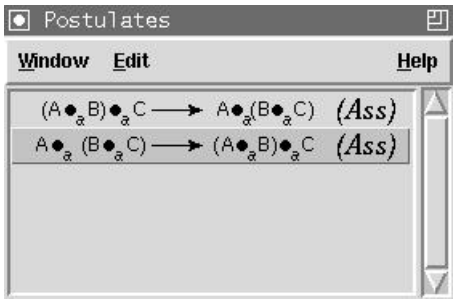

Figure A.22: Two postulates for associativity.

Note that Grail will complain if one of the following holds.

- There are multiple occurrences of a variable on either side of the postulate arrow.
- A variable occurs on only one side of the postulate arrow.
- A postulate has more occurrences of unary connectives on the left hands side than on the right hand side.

In the last case, you can overrule Grail's complaints and store the postulate anyway, though this might lead to nontermination of Grail's proof search mechanism.

Finally, if you try to store a postulate which is already a consequence of other postulates in you fragment, Grail will notify you of this.

Editing a Postulate To edit an existing postulate, you basically have the same options as for editing a lexical entry. You can cut, copy and paste by using [Edit/Cut], [Edit/Copy] and [Edit/Paste] respectively.

You can delete a postulate by selecting it and using [Edit/Delete Postulate] from the postulate window. A final option is to disable postulates. This makes a postulate unavailable without actually deleting it and is a way of experimenting with different sets of structural postulates to see which structural rules you really need.

Rewriting Now that we have added the structural postulates for associativity to the grammar, we can see if this finally makes the sentence 'Livia hates someone' derivable. Double-click again on this sentence from the main menu, select [Fail] after the first lexical lookup, then select [Leap] to generate the first acyclic, connected proof structure for this lookup and the rewrite window will appear, looking exactly as before in Figure A.16.

Now, however when we select the '$\circ_a$' node just below the '/' node, the popup menu will display we have the option here to use the 'Ass' structural rule we have just added. We select this from the menu, to produce the label shown in Figure A.18. Grail automatically keeps track of all alternatives, and if we change our mind about the current rewrite, we can select [Run/Undo!] from the menu or press <u>. If we select the '/' node from this configuration the popup menu will display 'Res', which indicates we can perform the residuation conversion for '/' and eliminate this node from the label, which gives the result shown in Figure A.23.

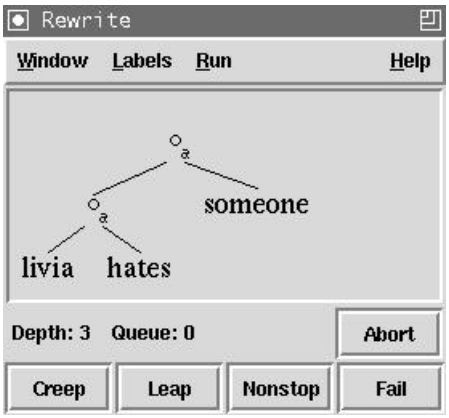

Figure A.23: Correct label for 'Livia hates someone'.

This is a correct label, but if you want you can apply an extra associativity step by selecting the top '$\circ_a$' node. When you are satisfied with the currect label, press either [Creep] or [Leap] and Grail will start producing the LaTeX output, then continue trying to find proofs for a different axiom linking.

A.3 Reference Guide

A.3.1 The Main Window

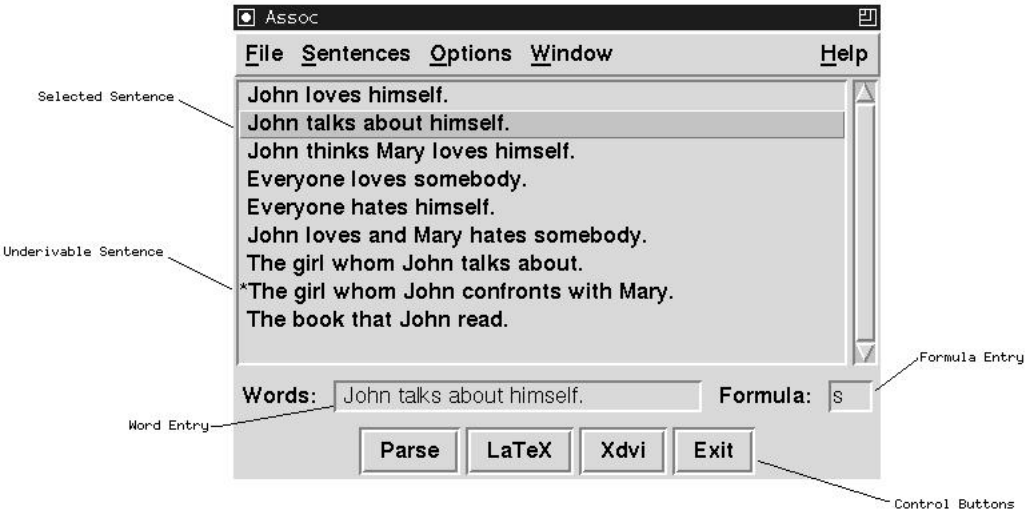

Figure A.24: The main window.

The main window is your interface to the theorem prover. From here you can parse grammatical expressions, load and save grammar fragments, and open other windows to view and edit these fragments.

The window will display a list of previously parsed sentences for the current fragment, and an input section where you can enter new expressions.

Clicking on one of the sentences will display the words in the word entry section, and the goal formula in the formula entry section. You can edit the words and the formula, and parse the sentence by pressing (Enter) or the [Parse] button. Double clicking one of the sentences will parse it immediately.

When parsing, a status window will appear which gives an indication of the computations being performed, and when ready will display either 'done' or 'failed' depending on whether a proof was found. The [Abort] button cancels the computation when you run out of patience.

Tokenization Grail will tokenize an input sentence in the following way. First of all, the following characters are, by default, defined as interpunction characters and will be treated as spaces.

```
! " ' - . : ; ? `
```

Grail contains a Prolog hook you can use to override the default behaviour. If your fragment contains a declaration of the form

```
special_string(String,Atom).
```

Grail will tokenize the String which is given as the first argument of `special_string/2` as the Prolog atom given as the second argument. Examples would be the following.

```
special_string("?",'?').
special_string("can't",'cannot').
```

Note that you still have to add lexical entries for '?' and 'cannot' if you want to use this in your fragments.

Currently, you can only add `special_string/2` declarations to your fragment by editing your file manually.

Command Buttons below the entry sections you will find the following command buttons.

[Parse] Parses the words in the input entry as a formula described in the formula entry. The output is sent to LaTeX.

[LaTeX] Sends the results of the previous parse to LaTeX; this is useful if you have changed some of the output options.

[Xdvi] Sends the result of the previous parse to LaTeX and displays them using the xdvi previewer.

[Exit] Exits the program.

Menu Bar from the menu bar of the main window, you can access the following operations.

[File]

[About] Prints information on the release date and version number.

[New Fragment] Starts a new grammar fragment from scratch. All previous information will be lost.

[Consult Fragment...] Loads a grammar fragment.

[Save Fragment...] Saves your fragment.

[Compile Prolog Source...] Compiles a Prolog file.

[Close] Iconifies the main window.

[Quit...] Hasta la vista, baby.

[Sentences]

[Clear Entry] Erases the words and formula.

[Parse] Parses the selected sentence.

[Delete] Deletes the selected sentence from the sentence list.

[Options]

[Prolog Messages] A choice between Quiet and Verbose. When set to Quiet, only some information about the time a computation takes will be sent to screen. When set to Verbose, a lot more information about the state of the computation will be printed. Defaults to Quiet.

[View Format] A selection of LaTeX output formats for the Grail natural deduction output. Current possibilities are 'none' (no LaTeX output, resulting in faster execution of Grail because it is unnecessary to keep track of the *path* to the solution), 'dvi', 'postscript' and 'pdf'. Defaults to dvi.

[Viewer Geometry] A choice of the window size of the LaTeX previewer. Possible values are 320x200, 640x400, 800x600, 1024x800 and 1280x1024. Defaults to 800x600. Note that some previewers ignore their geometry parameters.

[Natural Deduction Style] A choice between Prawitz style natural deduction and Fitch style natural deduction. Defaults to Prawitz.

[Proofs]

[Eta Long Proofs] When this checkbutton is on, eta long natural deduction proofs will be produced. Defaults to off.

[Hypothesis Scope] When this checkbutton is on, the scope of a hypothesis in Fitch style natural deduction will be indicated by a vertical bar. Defaults to on.

[Labels]

[Output Labels] When this checkbutton is on, labeled deduction proofs will be produced. Defaults to on.

[Implicit Structural Rules] Structural rule applications will be hidden.

[Collapsed Structural Rules] Successions of multiple structural rules will be collapsed into one.

[Explicit Structural Rules] Each structural rule is portrayed explicitly. This is the default setting.

[Formulas]

[Reduce Macros] When this checkbutton is on, complex formulas will be reduced by the macro definitions. Defaults to off.

[Semantics]

[Output Semantics] When this checkbutton is on, lambda term semantics will be printed with the formulas. Defaults to off.

[Functional Notation] Switches off the Montague-style notation conventions and displays complex function terms normally.

[Predicate Notation] Uses Montague-style notation conventions displaying a term like $((f\ y)\ x)$ as $f(x, y)$.

[Reduce Semantics] When this checkbutton is on, lambda term reductions will be performed whenever possible. Defaults to off.

[Substitute Lexical Semantics] Formulas will be assigned their lexical meaning recipes instead of semantic variables. Defaults to off.

[Semantics For Unary Connectives] When this checkbutton is off, the semantic constructors for the unary connectives will be ignored. Defaults to on.

[Colors...] Opens a color selection window allowing you to change the standard colors of the Grail application.

[Fonts...] Open a font selection window allowing you to change the font family, weight, slant, width and size. Font selection changes will only affect the proof net and the rewrite window.

[Save Current Options] Save the currect settings for all options to the file `.grail_default_options.pl` which will be automatically loaded the next time you start Grail.

[Restore Default Options] Return the options to their initial state, as if you had just restarted Grail.

[Window]

[Status/Proof Net Window] Opens the status or proof net window, depending on whether debugging is turned on or off. See sections A.3.2 and A.3.3.

[Rewrite Window] Opens the rewrite window, only available when debugging is turned on. See section A.3.4.

[Lexicon Window] Opens the lexicon window. See section A.3.5.

[Postulate Window] Opens the postulate window. See section A.3.7.

[Analysis Window] Opens the analysis window. See section A.3.9.

[Help]

[On This Window] Gives a help message.

A.3.2 The Status Window

The status window gives information about the current state of the computation, and allows you to abort time-consuming parses. It will open automatically during proof search, or you can open it by selecting **[Window/Status Window]** or by typing (Control-s).

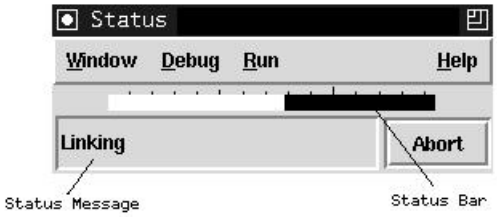

Figure A.25: The status window.

The white part of the status bar gives an estimate of the number of links which have not been tried yet.

The status message can be one of the following.

Initializing Garbage collecting, preprocessing.

Linking Performing axiom links.

Rewriting Performing label conversions.

Generating Output Generating LaTeX output, and sending it to a file.

Done Computation terminated, one or more derivations were found.

Failed Computation terminated, no derivations were found.

Aborted User got bored and pressed the [Abort] button. If derivations were found, they can still be viewed.

Menu Bar From the menu bar, we can select the following.

[Window]

[Close] Iconifies this window.

[Debug]

[Automatic] Debugging off. Grail will search for proofs without user guidance.

[Interactive] Switches on the interactive debugger.

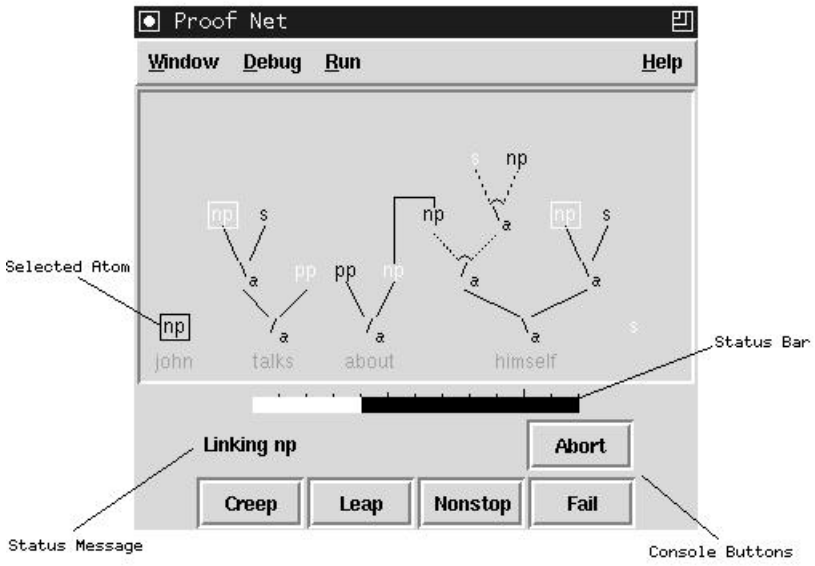

Figure A.26: The proof net window.

A.3.3 The Proof Net Window

When the interactive debugger is on, the status window will be replaced by the proof net window. In the proof net window we see the current partial proof structure, with the decomposition trees of the formulas the current lookup assigns to the words from the sentence above the corresponding word. Positive atomic formulas are drawn in white and negative atomic formulas drawn in black. Here atomic formulas of opposite polarity are linked until we find a proof structure which is both acyclic and connected.

The console buttons offer the following options.

[Creep] Will perform the next step in the computation, then wait for interaction.

[Leap] Will return after a total linking for the current lookup has been found or to the next lookup if no such linking exists.

[Nonstop] Will perform the rest of the proof search automatically.

[Fail] Will abandon the current branch of the search space and continue with the next untried branch.

[Abort] Aborts the current proof attempt.

In addition, you can click on the atomic formulas themselves to have complete control over the order in which the axioms are linked. As a first step you select any atom not currently linked by an axiom link. The selected formula will then appear in a black box and the atoms of opposite polarity which have not been tried before will appear in a white box, as shown in Figure A.26 on the facing page. You can then click any of the boxed formulas to perform an axiom link.

In addition, if you know you are only interested in one specific choice of the possible axiom links, you can keep the `Control` key depressed when you press the mouse button in order to *commit* yourself to a specific axiom link. This is equivalent to first selecting every other possibility followed by pressing the [Fail] console button.

If at any time you perform a link which results in a cyclic or disconnected proof structure, you will get a message and the current link will fail.

Be warned that by selecting [Fail], [Abort] or using the commit option, you cut part of the search space and may miss valid proofs if you are not careful.

Menu Bar For the menu bar, we can select the following.

[Window]

[Save Postscript] Saves the current (partial) proof structure to a postscript file.

[Close] Iconifies this window.

[Debug]

[Automatic] Debugging off. Grail will search for proofs without user guidance.

[Interactive] Switches on the interactive debugger.

[Run] Setting this option to [Nonstop] will cause Grail perform all axiom links without user interaction. Defaults to [Creep].

A.3.4 The Rewrite Window

When the interactive debugger is on, you can open the rewrite window by selecting **[Window/Rewrite Window]** or by typing (Control-r).

The rewrite window (Figure A.27 on the next page) displays the current label and allows you to perform rewrite operations on this label. Clicking on a node of the label will cause a pop-up menu with the label conversions rooted at that node to appear. You can apply a conversion by selecting it from the menu. Any alternatives to your choice will be added to the queue.

The status message gives you an indication of the number of unvisited labels in the queue and of the current depth.

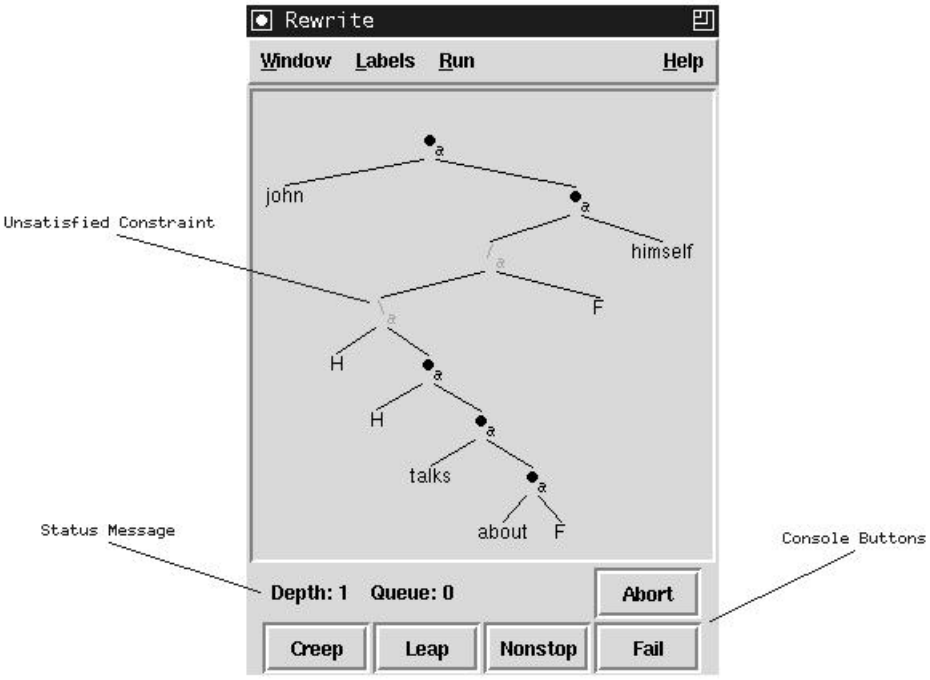

Figure A.27: The rewrite window.

As shown in the figure, some label constructors are drawn in dark grey. These correspond to unsatisfied constraints, which are checked by the label conversions shown in Figure A.28.

For the non-associative base logic, these are all available conversions. However, you can relax the constraints by specifying your own structural postulates as specified in section A.3.7. Each structural postulate can be applied backwards as a label conversion.

You can rewrite a label until you reach one where all constraints have been satisfied and the words are in the order required by the input sentence. Grail will only check if you meet these conditions when you press the [Creep] or [Leap] button in order to allow you to continue rewriting a label even if all constraints have been satisfied.

The console buttons offer the following options.

[Creep] Will add all one step conversions from the current label to the back of the queue, then continue with the first element of the queue.

[Leap] Will return only after all label constraints have been satisfied.

[Nonstop] Will perform the rest of the proof search automatically.

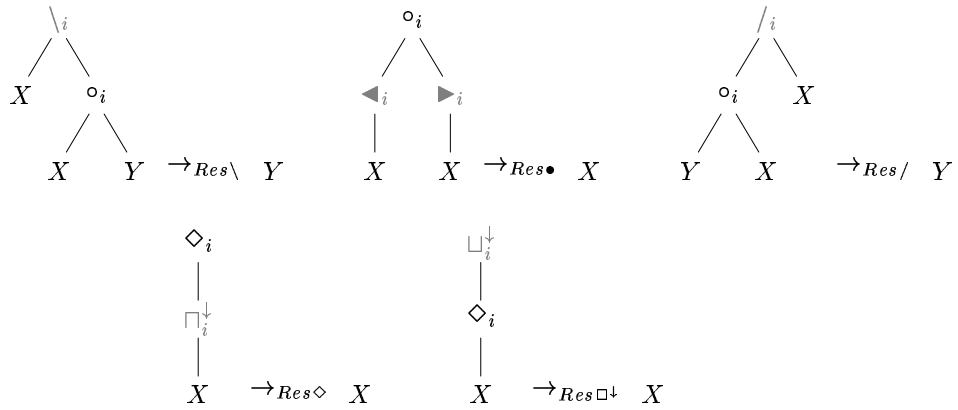

Figure A.28: Residuation conversions

[Fail] Will abandon the current branch of the search space and continue with the next item on the queue.

[Abort] Aborts the current proof attempt.

Menu Bar For the menu bar, we can select the following.

[Window]

[Postulate Window] Opens the postulate window.

[Save Postscript] Saves the current label to a postscript file.

[Close] Closes this window.

[Labels]

[No Eager Evaluation] Will prevent Grail from doing any early failure on label conditions.

[Automatic Eager Evaluation] Will cause Grail to perform automatic eager label conversions. This is the default.

[Manual Eager Evaluation] Will allow the user to perform eager label conversions himself. Be careful, as careless eager conversions may prevent solutions from being found.

[Run] Setting this option to [Nonstop] will cause Grail perform all label conversions without user interaction. Defaults to [Creep].

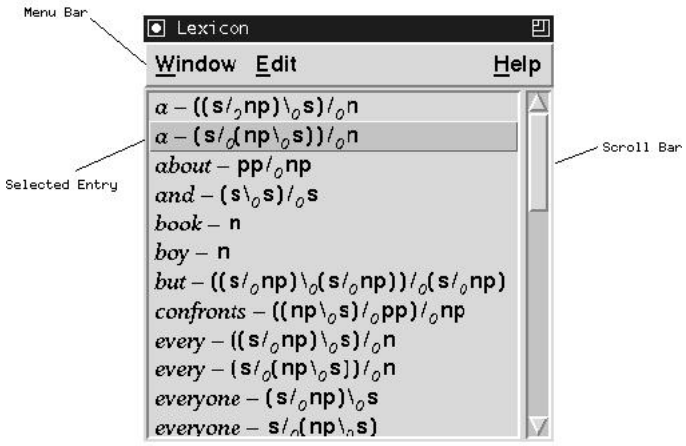

Figure A.29: The lexicon window.

A.3.5 The Lexicon Window

You can open the lexicon window from the menu bar in the main window by selecting **[Window/Lexicon Window]**, or by typing (Control-l).

The lexicon window (Figure A.29) displays a list of the words in the fragment and of the formulas assigned to them. From here you can edit, delete or enter new lexical entries.

Clicking an entry will select it, indicated by the selection bar. The next edit or delete command will then be applied to that entry.

Double-clicking one of the entries will open the edit lexical entry window, with that entry displayed in it (see section A.3.6 for more on the editing of lexical entries).

Clicking one of the entries with the (Control) key depressed will delete it.

Menu Bar from the menu bar of the lexicon window the following operations are available.

[Window]

[Close] Closes the lexicon window.

[Edit]

[New Entry] Opens the edit entry window.

[Edit Entry] Shows the selected lexical entry in the edit entry window.

[Delete Entry] Deletes the selected lexical entry.

[Help]

[On This Window] Gives a help message.

A.3.6 Editing a Lexical Entry

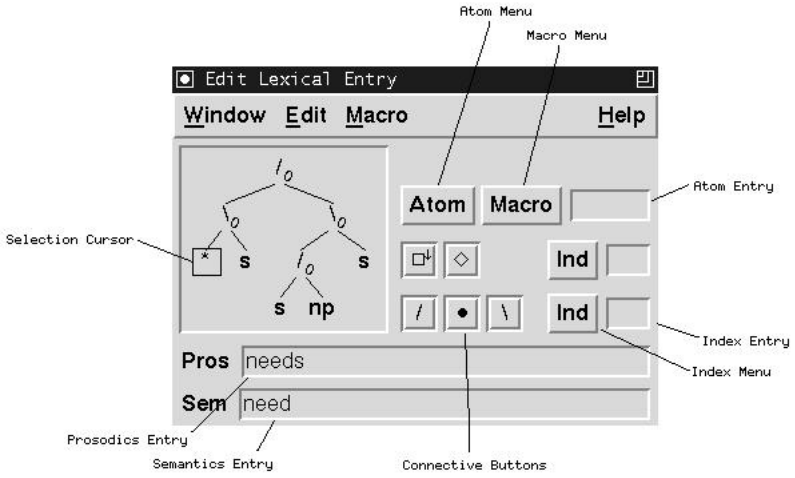

Figure A.30: The edit entry window.

The edit entry window (Figure A.30) is where you modify existing entries in the lexicon or create new entries from scratch. You can open the edit entry window by selecting **[Edit/New Entry]** or **[Edit/Edit Entry]** from the lexicon window.

The edits you perform here will only be stored in the lexicon when you press (Control-s) or select **[Edit/Store Entry]** from the menu bar, so you don't have to worry about accidentally modifying your lexicon.

A lexical entry consists of three parts: *prosodics*, a syntactic *formula* and *semantics*.

Formula the formula edit fields take up the upper section of the window.

Selection The formula is displayed as its construction tree. You can select a part of the formula by clicking on it. The selection cursor appears as a box surrounding the root of the selected tree.

Insertion Points A special constant '*' functions as an insertion point in the formula. It is not a part of the formula language. By pressing (Control-k) the selected tree will be replaced by this constant, and copied to the paste buffer.

When an insertion point is selected (as shown in Figure A.30), you can insert something at that position in one of the following ways.

[Paste] Pressing (Control-y) will insert the contents of the paste buffer to this position.

[Atom] By clicking on the atom menu, you can insert one of the atomic formulas found in this fragment. Alternately, you can type in a new atom in the atom entry, followed by (Enter). Atoms should start with a lower case letter, and be followed by any number of alphanumeric characters or _.

If you want to use complex Prolog terms as atomic formulas, you will have to explicitly declare them in your fragment file. For example by using the following.

```
atomic_formula(np(nom)).
atomic_formula(np(acc)).
```

Note that you can currently do this only by editing your fragment manually.

[Macro] A very simple macro facility is provided, where you can give a name to commonly occurring formulas. Selecting one of the macros from the macro menu will insert it at the current position.

[Constructor] You can insert a unary or binary constructor by selecting an index from the index menu next to the buttons for these constructors (or typing in a new index in the index entry next to it) and pressing the button for the connective you wish to insert.

Prosodics The prosodics of an entry is the way it will appear in your expressions. You can enter a Prolog term in the prosodics entry section. The current version does not support lexical entries consisting of more than one word.

Semantics You can give your lexical entry a Montague-style meaning recipe in the semantics entry section. Editing the semantics in the current version is very cumbersome, as it requires you to type in the internal semantic representation. It is recommended you leave the semantics field empty or type in a single constant. If you really want to enter lambda term meaning recipes you can use Table A.1 on the facing page to convert lambda terms to Prolog terms.

Menu Bar In the edit lexical entry window, you can select the following from the menu bar

[Window]

[Close] Closes this window.

Lambda Term	Prolog Term
variable	Prolog variable
constant	Prolog constant
$(f\ x)$	`appl(F,X)`
$\lambda x.t$	`lambda(X,T)`
$\langle x, y \rangle$	`pair(X,Y)`
$\pi^1 x$	`fst(X)`
$\pi^2 x$	`snd(X)`
$^\vee t$	`debox(T)`
$^\wedge t$	`conbox(T)`
$^\cup t$	`dedia(T)`
$^\cap t$	`condia(T)`
$\neg x$	`not(X)`
$x \wedge y$	`bool(X,&,Y)`
$x \vee y$	`bool(X,\/,Y)`
$x \rightarrow y$	`bool(X,->,Y)`
$\forall x.t$	`quant(forall,X,T)`
$\exists x.t$	`quant(exists,X,T)`
$\iota x.t$	`quant(iota,X,Y)`

Table A.1: Representation of semantic terms in Prolog

[Edit]

[Clear Entry] Erases the formula, prosodics and semantics fields.

[Store Entry] Stores the current lexical entry in the lexicon.

[Cut] Cuts the current selection to the paste buffer.

[Copy] Copies the current selection to the paste buffer.

[Paste] Pastes the buffer to the current position.

[Macro]

[Store Entry As Macro] Stores the formula of the current entry as a macro. The macro will take its name from the atom entry field.

[Store Selection As Macro] Stores the selection as a macro. The macro will take its name from the atom entry field.

[Help]

[On This Window] Prints a help message.

A.3.7 The Postulate Window

The postulate window (Figure A.31 on the next page) displays the structural postulates in the current fragment. From here you can delete or edit structural postulates.

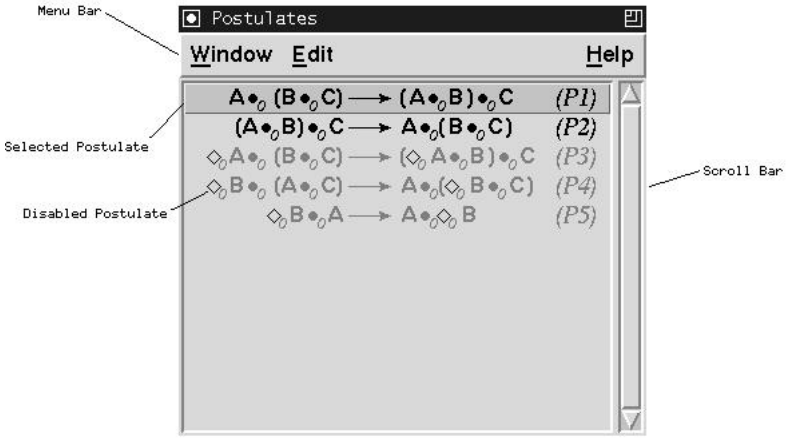

Figure A.31: The postulate window.

You can open the postulate window from the menu bar in the main window by selecting **[Window/Postulate Window]**, or by typing (Control-p).

Clicking on a postulate will select it. This will cause a selection bar to appear over it, and allows you to perform the operations in the edit menu on it.

Double clicking a postulate will display that postulate in the edit postulate window.

Clicking a postulate with (Control) depressed will delete that postulate.

Pressing mouse button 2 over a postulate will change the status of the postulate from enabled to disabled or vice versa. This allows you to experiment with the effects of structural postulates without having to create several versions of the same fragment.

Menu Bar From the menu bar, the following options are available.

[Window]

[Close] Closes the postulate window.

[Edit]

[New Entry] Opens the edit entry window.

[Edit Entry] Shows the selected structural postulate in the edit postulate window.

[Delete Entry] Deletes the selected structural postulate.

[Disable/Enable Postulate] Toggles the selected postulate between enabled and disabled.

[Help]

[On This Window] Gives a help message.

A.3.8 Editing a Postulate

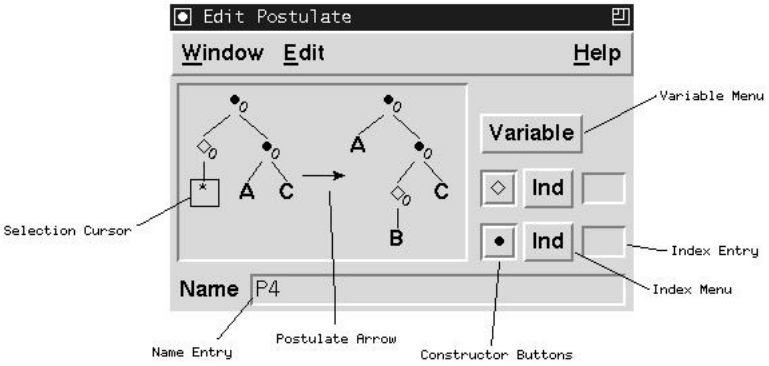

Figure A.32: The postulate edit window.

Editing a postulate is much like editing a formula. There is now a formula on both the left and the right hand side of the postulate arrow. Selection and cut/copy/paste can be performed as before.

Instead of atomic formulas we now have structural variables, which can be inserted from the variable menu, and our choice of constructors is limited to $\diamond$ and $\bullet$.

Postulate Arrow By clicking on the postulate arrow it will change from $\rightarrow$ to $\leftrightarrow$ to $\leftarrow$. This makes it easier to store equivalences or inverses of postulates. In the postulate window, all postulates will appear in their left to right version regardless of the postulate arrow, so storing a postulate $A \leftrightarrow B$ will in fact be the same as storing both $A \rightarrow B$ and $B \rightarrow A$.

Postulate Names You can give a postulate any name which is printable in LaTeX math mode.

Valid Postulates The computational architecture poses some limitations on the type of postulates allowed in your fragments. Grail will report an error when you try to store postulates of the following form.

- There are multiple occurrences of a variable on either side of the postulate arrow.

– A variable occurs on only one side of the postulate arrow.

In addition, because of the backward chaining proof search strategy, a warning will be generated when a postulate has more constructors on the left hand side than on the right hand side. If you add one of these postulates to your fragment, the proof search algorithm is not guaranteed to terminate. This is the same restriction discussed in Section 8.2 and ensures that proof search is PSPACE complete as opposed to (potentially) undecidable.

Menu Bar the menu bar allows you to access the following functions.

[Window]

[Close] Closes the edit postulate window.

[Edit]

[Clear Postulate] Erases the postulate in this window.

[Store Postulate] Stores the postulate in memory. It will now appear in the postulate window.

[Reverse Postulate] Swaps the left and right hand sides of the postulate.

[Cut] Deletes the selected part of the postulate, and copies it to the paste buffer.

[Copy] Copies the selected part of the postulate to the paste buffer.

[Paste] Pastes the contents of the buffer to the place of the selected variable.

[Help]

[On This Window] Hmmm, what does this window do?

A.3.9 The Analysis Window

The analysis window (see Figure A.33 on the facing page) is where you can improve the performance of the theorem prover by setting the parameters for early failure. This can be done either automatically or by hand.

External Modes Sometimes you may want to prevent a mode from occurring in the output, because it is used only as a grammar internal or auxiliary mode. By default all modes will be external, but you can set modes to internal by turning off their checkbutton here.

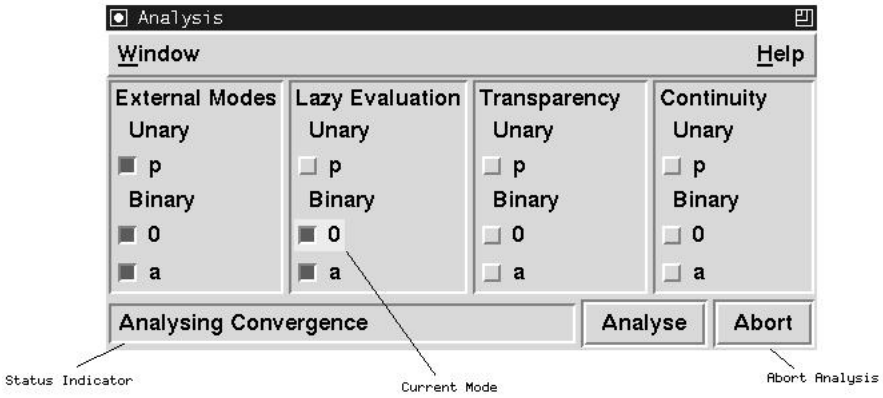

Figure A.33: The analysis window.

Lazy, Transparent and Continuous Modes Three forms of early failure are supported which apply only to structural postulates satisfying some criteria. See (Moot 1996) for descriptions of these criteria. All can be detected by the program, and only the lazy reductions test is expensive to compute. When the program suspects checking for lazy reductions will take up an unreasonable amount of time, you will get a choice to set these parameters to their default, safe settings and only perform the other tests.

Menu Bar From the menu bar, the following options are available.

[Window]

[Close] Closes the analysis window.

[Options]

[Show Status] Gives a description of Grail's estimate of the current analysis settings. This can be *manual* if the settings were performed by the user, *safe* if performance is perhaps not optimal but will not prevent solutions from being found, *optimal* if a complete analysis has been performed on the current postulates, or *unknown* if postulates were added after the last analysis.

[Analyse Postulates] Performs a complete analysis of the postulate set.

[Analyse Convergence] Will only check if the label reductions converge for eager evaluation. This is generally time-consuming.

[Analyse Transparency] Will only check if word order constraints can be applied eagerly.

[Analyse Continuity] Will only check for which modes continuity labeling applies.

[Safe Settings] Switches off *all* early failure.

A.4 Conclusions

We have given an overview of the Grail interactive theorem prover and its underlying logical theory. Grail displays an intuitive representation of the state of the computation and allows the user to guide the computation by interacting with this representation.

On the proof net level, an advantage over sequent or natural deduction systems is that linking atomic formulas is a relatively trivial way to generate all proofs for a given statement. User guidance allows more experienced users to perform the axiom links they are interested in immediately, thereby sidestepping the $O(n!)$ complexity.

On the label rewrite level, it is often enlightening to see Grail (ab)use your carefully chosen structural rules in unintended ways, showing linguistically incorrect predictions of your logical theory, or to see it fail to satisfy a critical constraint, pointing to a missing or not sufficiently general structural rule. User interaction can considerably improve the performance by allowing the user to perform the intended label conversions himself.

Finally, though proof nets are in many ways an optimal proof theory for proof *search*, as previous chapters ought to have shown, natural deduction is generally a better theory to *display* them. Therefore, source code which transforms the completed proof net into LaTeX natural deduction output is included with the release.