```
Use the HowTo? {1} button in MainMenu {2} to raise this object.    doc_for_how_to {1}

See "Notable Features of this Interface" {3}.

How to: Get Started Using this Interface {4}
        Find Documentation and Definitions {5}
        Make Hardcopy of Editor Documents. {6}
        Tailor this Editor for Yourself. {7}
        Use Paul Jackson's Nuprl Manual {8}
        Find Explanations of Primitives {9}

        Use the Active Mouse {10}
        Move the Cursor {11}
        View an Object by Name {12}
        Track Objects {13}
        Close a Window {14}
        Undo {15}
        Reveal Hidden Parts of a Term {16}
        Explode an Operator {17}

        Insert or Delete a Term at the Point {19}
        Edit Strings within Terms {20}
        Mix Editing Modes {22}
        Insert Comments {23}
        Edit ML Code {24}
        Edit Proofs {25}
        Edit WORDS {29}
        Edit Operator Definitions (ABS) {32}
        Edit Display Form Specs {34}
        Do Name Completion {41}
        Toggle between Cognate Forms {42}
        Regroup Term Sequences {31}
        Search for Terms in a Window {43}

        Create New Objects & Operator Defs {45}
        Execute ML Expressions {46}
        Find Library Objects {50}
        Make Object References {51}
        Make Buttons {53}
        Make Keyboard Commands from Buttons {54}
        Cut & Paste {56}
        Insert Constable's Dependent-Type Notation {57}
        Post your Changes to an Object {58}
---

ObTracker Viewer I/O Scratch NEW MAP LibSearch                         main_menu {2}
FIND {60} MISC ML WORDS PF
Buttons CmdReview HowTo? {1}
```

While these documentation pages do not constitute a tutorial or an exposition for the main ideas, one can get some idea of the capabilities from reading through them. Start at "How to use this interface" {1}.

Notable features:

1. This editor is powerful and extensible. Indeed, it was built by bootstrapping from the Nuprl editor primitives, through the creation of: ML code for both inference and editing utilities; operator definitions (used as macros); term-display methods; bindings of editor-primitive combinations to keyboard commands; documentations objects; menus of active button-like expressions; and examples. The ML code constitutes the bulk of these sources.

2. The structure of Terms used for proofs, program code, definitions, etc. is largely independent of how they are to be displayed. Display is user-specifiable and fairly flexible, exploiting a number of methods standard in informal notation. Although many operations on terms, such as making inferences, are mostly independent of terms' appearance, interactive structure-editing is highly influenced by how terms are displayed. That is, the editing operations are implicitly parameterized by various features of user-selected display, and editing is basically WYSIWYG. (And yet, the bulk of defined editing operations is still display-independent.)

3. While one could think of Nuprl Terms as parse trees generated from a very simple grammar, there would be little reason to do so, since, on one hand, the grammar is rather degenerate, and on the other, the strings one parsed would, it turns out, not be seen by the normal user; indeed, there is typically, though not necessarily, some user-specified ambiguity in the display of terms, so there is no certain path from the appearance to the underlying term. There is no significant "grammar" per se; any term can appear as an immediate subterm of any operator that takes subterms. One consequence is that rather than editing terms top-down or bottom-up, one may edit them "thought-first". Another is that operator definitions serve not only as mathematical notations, but also as system macros.

4. Naturally, there are hyperlinks {51} between objects of the system (the content of an object is expressed as a term, and objects have names).

5. The development of new material by imitation of existing material is facilitated in various ways. One finds something that approximates what one wants, then adapts it.

6. One can attach "actions {53}" to terms which are performed in response to various user gestures such as mouse-clicking on a term. In addition to a few editing operations, one can execute arbitrary user-written ML code {24} this way. Indeed, the bulk of this editor is written in ML.

7. From a "meaningful" non-primitive expression of the system one can normally track down the sources that give it its meaning, such as operator {5} definitions (macros), and definitions of ML globals {64} (in the standard "prelude"; in the editor; in user defined code). And from a term instance in the editor one can find the sources that make it display {34} that way.

8. Various methods have been incorporated for finding documentation {5} pertinent to a given term instance, such as what keyboard commands one might use to insert or modify them, or whether they are "action" terms and what they do.

9. Other key utilities include name completion {41} in several name spaces, finding theorems and definitions that mention {50} a given operator, finding theorems that explicitly cite {50} a given lemma, finding ML global definitions that cite {64} another given ML global identifier, and "in-place" typing of ML code, which is useful for understanding and correcting ML code piece by piece.

10. Informal material can be combined with formal material. Indeed, this documentation as well as that of Nuprl Basics was built that way with this editor.

11. The style of interaction is "on-the-fly". When a user creates or modifies source material, such as operator definitions, buttons, display methods, ML code for inference or utilities, it is immediately usable, and no "system rebuild" is required.

(Oct 2001 - sfa)

It is probably best to start using this editor simply by browsing the Nuprl library. Tips for editing may be developed later -- for now you'll just have to survey the body of documentation (see HowTo {1}).

Keyboard Command References.
Use ‘(mouseleft)’[324] on KbdCmdReferences {65}, for how to read command names in this documentation.

X-windows and the Lisp Window.
For advice about configuring your Xwindows setup for use in Nuprl, click on XwindowSetup {66}.

Quick Documentation and Viewing.
Here are mouse commands for telling you about operators you point at. In increasing degrees of detail: ‘(c-(mouseleft))’[310] ‘(cm-(mouseleft))’[327] ‘(cm-(mousemiddle))’[311] ‘(cm-(mouseright))’[312]

The Point.
It is important to master setting the "point", or Nuprl "cursor".
WARNING: This "point", is not the same as the Xwindow cursor that tracks the mouse; the point is simply a term in a Nuprl window indicated by highlighting; it may not even be in the same window as the Xwindow cursor.
It is recommended that you become fluent with the following commands:
‘(c-(mouseleft))’[310] ‘(m-(mouseleft))’[309] ‘(mousemiddle)’[313]
‘(c-(mousemiddle))’[314] ‘(mouseright)’[317]
See BasicMouseCmds. Scroll up and down with ‘{(UP)≡(c-u){(c-k)≡(cm-k)}}’[61] and ‘{(DOWN)≡(c-u)(c-j)}’[60].

The "Active" Mouse Button - ‘(mouseleft)’[324].
On normal terms, ‘(mouseleft)’[324] simply sets the cursor, but many terms are "buttons", and ‘(mouseleft)’[324] on one performs some function, often with respect to the Nuprl "point". To facilitate use of these buttons, the mouse-command ‘(cm-(mouseleft))’[327] will usually print a line of documentation about what would happen if you used ‘(mouseleft)’[324]. The simplest form of button is VIEW→ [ob name] which views the named object. See ActiveMouse {10}

Raising Windows with ‘show’[55] or ‘(cm-(mouseright))’[312].
The Scratch button in the VIEW→ main_menu {2} will raise a new empty comment object.
The ‘show’[55] command will normally raise a definition, proof, or documentation object determined by the point. You can can also both set the point and do a ‘show’[55] with the single command ‘(cm-(mouseright))’[312].
See ProofTerms {67} for explanation of how proofs are displayed. You can start browsing part of the standard library by clicking HERE. See ObjectViewing {12}.

Closing a Nuprl Window with ‘(c-q)’[286] and ‘(c-a)(c-q)’[289].
‘(c-q)’[286] will close windows that have not been altered since being raised, and ‘(c-a)(c-q)’[289] will force a window to close, altered or not -- take care.

To summarily quit Nuprl, ctrl-C Lisp then, if it's Lucid Lisp, (quit), if Allegro, :exit.

How to find Documentation and Definitions.   Review pertinent commands? YES

Miscellaneous Operator Documentation.

The ‘show’[55] command will show a definition of some kind, or object pointed at. ‘(cm-(mouseright))’[312] will do this wherever you click.

The ‘(c-a)(c-?)’[28] command will glean information from the system about the point-term; mostly about the operator. ‘(cm-(mousemiddle))’[311] will do this wherever you click.
It will give you a pointer to the ABS and DISP objects for the point-term, and perhaps comments taken from those objects.
It will tell you if the toggling commands ‘(backspace)’[99] and ‘(c-a)(c-(backspace))’[101] have any effect, and describe the effects of toggling terms like it. VIEW→ doc_for_toggling {42}
It will also list commands that probably insert the operator.
If the point-term is a (cached) global ML-id, then this will be mentioned, and if an appropriate comment can be gleaned from the definition, it will be shown.
If the point-term is an edit command, then it will also be "reviewed" (see below).

Button Documentation.

Edit buttons are used by clicking on them with ‘(mouseleft)’[324]. There are a few special purpose buttons, but most buttons either view an object, or insert a term, or invoke ML (usually on the point).
There is an edit button, Buttons, which you will also find in MainMenu {2}, that will scan the point term for edit buttons and pop up a window revealing them, and any documentation it could glean from the system.
Here are some edit buttons you should see in MainMenu {2} to help find documentation; try the Buttons button on them:

  Buttons CmdReview HowTo? {1}

Edit Command Documentation.

Commands are usually displayed in a way that shows what you need to key in to invoke it. For example, here is a command that will move the point up to the parent: ‘(m-k)’[57]. For
You can get a "review" of a command by using the ‘show’[55] command on it. To review all commands, or a special class of them, use the CmdReview button which you can find in MainMenu {2}. VIEW→ doc_for_edit_cmd_review {69}
You can also get a review of a command keying in the command string immediately preceded by ‘(m-q)’[7].

Hypothetical Execution Documentation.

Many commands are rather generic, but are tailored for special behavior in different circumstances. For some of these there is a "hypothetical" version that simply advises the user in the LISP window as to what action would occur.

```
hypothetical ‘(mouseleft)’[324] is ‘(cm-(mouseleft))’[327]
hypothetical ‘(c-z)’[290]       is ‘{(c-n)≡(m-n)}(c-z)’[307]
hypothetical ‘(c-d)’[81]        is ‘{(c-n)≡(m-n)}(c-d)’[308]
```

ML Documentation (including Tactics). VIEW→ doc_for_ml_doc {64}

WWW page for Nuprl: http://www.cs.cornell.edu/Info/Projects/NuPrl/

The documents explaining this editor are meant for interactive use, but you may want offline hardcopy, especially if you are a new or prospective user.

(Because the command names and display forms for various terms will differ from user to user, these documents must be printed from within the Nuprl session where the command bindings and display forms are defined.)

Clicking on the button below with ‘(mouseleft)’[324] will write the tex file ~/nuprl-home/main_doc_obs.tex .

This button will also print a copy of all the individual keyboard command review documents, putting them in
~/nuprl-home/complete_kbd_cmd_review.tex
These commands will be numbered in the document, and references to these commands from either within the reveiw documents or from the other documents printed here will show these numbers along with the command.

```
   Click HERE to write documentation files.

   SHELL latex complete_kbd_cmd_review {~/nuprl-home}
   SHELL dvips -f complete_kbd_cmd_review.dvi
               > complete_kbd_cmd_review.ps
         {~/nuprl-home}

   SHELL ghostview complete_kbd_cmd_review.ps & {~/nuprl-home}
   SHELL lpr complete_kbd_cmd_review.ps {~/nuprl-home}

   SHELL latex main_doc_obs {~/nuprl-home}
   SHELL dvips -f main_doc_obs.dvi
               > main_doc_obs.ps
         {~/nuprl-home}

   SHELL ghostview main_doc_obs.ps & {~/nuprl-home}
   SHELL lpr main_doc_obs.ps {~/nuprl-home}
```

Tailoring this Editor for the User. QUIT

The user can specialize various aspects of this editor by writing some files in ~/nuprl-home/. These files, if they exist, are loaded, in the order they are listed below, when Nuprl is started with the first evaluation in LISP of (nuprl).

~/nuprl-home/nuprl_init
should contain an ML expression to be executed when session is started.
To review the contents of this file: REVIEW

~/nuprl-home/main_menu
should contain as its single object the user's main menu, if it is to differ from the default. To save your current one after updating, posting, and closing:
SAVE main menu

~/nuprl-home/kbd
should contain the objects defining the user's keyboard command binding set. You can change these bindings by doing a command review via the button CmdReview. The current collection can be saved by clicking on the following button:
SAVE kbd cmds

~/nuprl-home/redisplay
should contain the user's preferred versions of DISP objects to be used instead of those coming from the files he has loaded or plans to load.
You can move a DISP object into this set with this button:
Move DISP to Redisplay Area.
You can save the current set to this file with this button:
SAVE redisplay objects. ;;

~/nuprl-home/omit_as_cascade_ancestors_init
should indicate the user's preference for which operators should initially be omitted from definition cascades. You can save the current preference with this button: SAVE omissions

~/nuprl-home/thm_viewing_method_init
should indicate the user's default preference for viewing THM objects. You can save the current preference with this button: SAVE THM viewing preference ;;

~/nuprl-home/overwriting_init_load
should contain objects the user wants to load at startup, even overwriting previously loaded objects. Restraint is advised. Here is a button that will raise a worksheet for you to fill in: Set Initial Overwrites

~/nuprl-home/users_edit_cmd_defs
should contain an ML expr defining user-defined edit commands.
To review the contents of this file: REVIEW

doc_for_using_pauls_manual {8}

Paul Jackson's Manual for Nuprl (postscript) is the best documentation for the system, but it is integrated with an explanation of a significantly different editing environment.

Some significant differences:

The commands are quite different.

There is little support in this editor for sequences of characters with embedded terms; there is much more support for "structured" terms, especially for ML.

There is no ML top-loop. ML is easily executed from within any object.

---

doc_for_math_expression {9}

Finding Explanations of Mathematical Expression in Nuprl

There is a collection of documents explaining the basic concepts and methods used for mathematical expression in Nuprl. These were designed mainly to be read on the web, but are also accessible within this editor.

There is an index, Nuprl Basics.

Further, the 'show'[55] and '(cm-(mouseright))'[312] commands will raise relevant documentation when applied to instances of basic operators, for example on "lhs == rhs", "∃x:A. P(x)", "f(x)", "x+8" or "𝔹".

The behavior of these commands is parameterized with respect to the document in which they are executed, and when there are several layers of explanation, ultimately down to the primitive declaration of operators, one can chase them down by successive applications.

doc_for_active_mouse {10}

There are two main uses for what we may call the active mouse, ‘(mouseleft)’[324]. It simply sets the point cursor on ordinary terms such as mathematical or programming expressions.

But on a class of terms we consider "buttons" some action is taken, such as viewing an object, inserting a term or operator, or executing an ML expression. See Buttons. {53}

You can usually get advice on what ‘(mouseleft)’[324] will do by using ‘(cm-(mouseleft))’[327] instead, which may be thought of as a hypothetical ‘(mouseleft)’[324] command.

To see a list of documents pertaining to ‘(mouseleft)’[324], click HERE.

‘(mouseleft)’[324] on QUIT or ForceQUIT

<*>

will quit the object without posting or checking for change.

Safe alternatives to ‘(mouseleft)’[324] for cursor motion are
‘(c-(mouseleft))’[310] which puts the point on a term, and
‘(m-(mouseleft))’[309] which puts the point in text if possible.

Point, Focus, Zoom doc_for_point_focus {11}

For a full review of relevant commands, use ‘(mouseleft)’[324] on: FullReview

To review a serviceable subset: ShortReview

Each open object has a local "point" cursor, and you can focus on one object at a time. Most edit commands operate on the point of the current object. The point will either be a term occurrence or a position within a text part of a term; there is also something called a screen position, but this editing setup doesn’t fully support it.

The point will only be shown in the "current" open window. If the point is on a term, the term will be highlighted; if it is in a text position, a thin vertical rule appears at the point where text would be inserted by normal commands.

To simply shift the FOCUS to the point of another open window you can use ‘(mousemiddle)’ anywhere on the target window.

You can also ZOOM on a subterm and unzoom. This is especially important when you need to examine terms elided because of insufficient space. For example, here the term ... is elided, and you can zoom on it by clicking on it with ‘(mouseright)’[317]. You can also zoom on other terms in case you want to focus your attention better or you want to speed up the layout of the display.

It is also sometimes necessary to SCROLL up or down when the window is too short for the term in the object.
(‘{(DOWN)≡(c-u)(c-j)}’[60] and ‘{(UP)≡(c-u){(c-k)≡(cm-k)}}’[61])

VIEW→ doc_for_string_editing {20} tells how to move the text-cursor.

While there are various ways to view objects, the basic one is by '(mouseleft)'[324] or 'show'[55] on what we call an object reference, or obref.

The basic form of obref is VIEW→ [ob name] which points to the object named in the text slot.

To insert this form of obref just use '{obr≡obref}'[133] on an empty term slot, then enter the name of the object in the text slot.
To simplify this task you can enter a few letters from the start of the object name and the use the name completion command '(c-&)'[23].

Note: If you use letters only of lower case, and omit the "_" character, then even objects whose name use upper case letters and have "_" inserted are included in the name completion search.

For more detail see:

Object References {51}
Name Completion {41}

The object tracking window, ObTracker, lists and categorizes objects that are open, recently opened, recently closed or marked as being of persistent interest. The window is refreshed by locating the point on the whole term, say with '(mouseleft)'[324] in the white space outside the displayed text, or by using '(c-z)'[290] from anywhere inside the window. There should be a pointer to ObTracker in the MainMenu {2}.

Here's what the character prefixes mean in ObTracker.

$^{+}$ indicates a currently open object.

* indicates an object Notice by the user,
as of persistent interest. (Forget will undo this.)

? indicates an object name that no longer names any object.

$^{\mathrm{o}}$ indicates a recently opened object.
They are tracked for a bit after closing.
You can change how many count as recent.

Number of recently opened obs to show: 3 Action: Increase, Decrease

Other objects will also be tracked for a bit after closing.

Number of recently closed obs to show: 6 Action: Increase, Decrease

The PRIORITY← group lists all the Notice (*) and recently opened objects. More recently Notice gets first listing, and after that more recently opened. Twice as many recently opened objects are listed in this group as are labelled $^{\mathrm{o}}$.

Utility buttons.

Recent	opens a listing of recently opened objects, more than are normally shown in the PRIORITY← group.
Viewer	raises the object that is considered to have opened the current object.
Notice	marks an object as being of persistent interest to the tracker.
Forget	unmarks the Notice mark.
Cycle-Q	cycles through open objects attempting safe quits. VIEW→ doc_for_how_to_quit_objects {14}
ClearHistory	Cuts down history of viewed objects to those that are currently open.
LibSearch	raises a worksheet for searching the library. Some ML editing is required. VIEW→ doc_for_ML_editing {24}

The utility buttons in the ObTracker come from →ob_tracking_aux2 and →ob_tracking_aux3.

doc_for_how_to_quit_objects {14}

The command ‘(c-a)(c-q)’[289] will unconditionally quit the object of current focus.

The command ‘(c-q)’[286] will normally attempt a safe quit, which means the quit only occurs if the object contents on display actually match the contents in the library. Changes to objects are buffered from the library, and must be posted to have effect.
VIEW→ doc_for_posting_objects {58}

The Cycle-Q button will attempt a safe quit ‘(c-q)’[286] on most open objects.

Only relevant if you're using the old "RED" proof editor: ‘(c-q)’[286] will also unconditionally quit an empty RULE box.

How to Undo Changes to an Object. doc_for_undo {15}

When an object is opened, the window is a sort of editing buffer, which must be posted in order to change the library object itself.
VIEW→ doc_for_posting_objects {58}

There is also a short list of the last few edit states maintained for each open object, which can be used to undo changes to the window. HERE for detail.

Further, since most destructive edit commands save the term before modified, you can often use the ‘(c-u)(c-u)’[83] command to restore the term, and save the accidentally created term temporarily. HERE for detail.

For more drastic changes, you can copy the current value of the object to the window with the ‘refreshme’[288] command, and the primitive command (m-x)restore will usually restore the window to the value it had when opened.

---

doc_for_display_suppression {16}

More than one display form may be usable for the same term.

It is possible to temporarily disqualify, or "suppress", a display form from being used, in order to fall back to another one that is preferred at that time by the user for some reason. The most common reason is to change from a form that hides parts of the term to one that reveals those parts.

There are three "scopes" of suppression available:

1. Global suppression disqualifies the display form throughout the system.
   ‘g(c-s)’[37] Suppress this display form globally.
   ‘g{(c-n)≡(m-n)}(c-s)’[38] Undo global suppression for this operator.
   ‘g(c-a){(c-n)≡(m-n)}(c-s)’[41] Undo all global suppression.

2. Window-local suppression is managed for each window independently. It suppresses the display form throughout the window. Because this kind of suppression is often very temporary indeed, and since complete refresh of a window can be slow, this suppression will not always be reflected except in the point until something precipitates the complete refresh, such as the user issuing a ‘(c-u)(c-l)’ command. (It would also be desirable to have an "occurrence specific" scope for suppression, but this was not built in. Perhaps later.)

‘(c-s)’[31]	Suppress the current display form in this window.
‘(c-a)(c-a)(c-s)’[30]	Repeatedly suppress until none are left.
‘{(c-n)≡(m-n)}(c-s)’[32]	Undo all suppressions local to this window for this operator.
‘(c-a){(c-n)≡(m-n)}(c-s)’[36]	Undo all suppressions local to this window.

3. Point suppression is an automatic suppression that occurs only at the edit point. It occurs when three conditions coincide:
   1. The point is located on the term occurrence in question.
   2. The display form has been given the attribute NotAtPoint.
   3. The point-suppression enable bit is set. This bit is an attribute of the window, so you can have it enabled in one window and disabled in another.

‘(c-u)(c-s)’[33]	Toggle the point suppression enable/disable bit.
‘(m-s)’[34]	If point suppression is disabled, enable it. Else, if a display form is actually being point-suppressed, suppress that display form throughout this window. (this is just so the user doesn’t have to disable point suppression to get at the point-suppressed display form.) Else just suppress this display form throughout this window.
‘{(c-n)≡(m-n)}(m-s)’[35]	Like ‘{(c-n)≡(m-n)}(c-s)’[32], but also disables point-suppression.
‘g(m-s)’[39]	Global version of ‘(m-s)’[34]
‘g{(c-n)≡(m-n)}(m-s)’[40]	Global version of ‘{(c-n)≡(m-n)}(m-s)’[35].

The point-suppression enable/disable bit is also used to control a few other automatic point-modification features. So if you are having a temporary problem with some automatic point modification happening, try disabling point-suppression.

Exploding a term generates another term that represents the main operator in a way that makes the operator structure explicitly viewable and editable.
Use '(c-a)(c-s)'[43] to explode or implode the point.

For example, exploding the term ∀x:A. B(x) gives
  IMPLODE→ all(A; x.B(x))
which shows the opid, and the subterms with bindings.
Exploding the term 12 gives IMPLODE→ natural_number{12:n}, showing its opid, and revealing that it has no subterms, and that it has a single n-type text slot.

The display forms for the exploded term representation elide empty sequences of components, such as empty parameter sequences, term sequences, and binding var sequences.
Use '(c-s)'[31] to reveal such elisions.

The main uses of explode forms are:

1. temporarily revealing the operator structure of a term whose display form is unfamiliar
2. getting access to the opid as text for use in editing
3. making a term-matching pattern to be used with an ML utility described in VIEW→ doc_for_edit_match {18}
4. modifying the operator in simple ways such as changing the opid, and permuting or deleting components; for other operator structure modification, as well as for creation of an operator for which you have no sample, the commands for "adjusting" operators are handier. See 'absadj'[120], 'adj'[125], and 'newop'[122].

Usually '(mouseleft)'[324] will implode it as well, which is why the mnemonic "IMPLODE" is used in the display. ML contexts are an exception since the explode form is often used in ML code in term-matching patterns, or as part of a crude quotation device; and since explode forms may be nested to form a term-matching pattern, being inside an explode form is also an exception. VIEW→ doc_for_edit_match {18}

You can get a palette of editing buttons appropriate to the explode form with the command 'expledit'[282], or '{ed≡edit}'[271] if you are already pointing to an explode form. The palette will be inserted as a cover.

This function is designed as a utility for matching against terms using patterns that are easy to create, using the editor, by modifying samples and variants of the terms you want to match.

The main match methods are:

1. Patterns may be nested, and a term is a pattern matching itself.
2. Edit slots are used as wild cards.
3. The explode form is used to specify patterns that match terms with non-homogeneous structure, such as various arities, opids, and variations on parameter structure and value. Edit slots in the exploded form that expect a sequence of structures actually match arbitrary sequences of such structures.
   VIEW→ doc_for_explode_form {17}
4. There is a disjuntion expression <pattern>|<pattern> that can be used freely on any part of a pattern (including opid and binding variables in the exploded form).
5. There is a *RECURSE* pattern that may be used within the pattern to match against a term recursively.
6. There is a pattern [string]:* which can match against any parameter type.
7. There are patterns [string]:substr, [string]:prefix, and [string]:suffix, for matching against substrings, prefixes, and suffixes of parameter values and opids.
8. As an abbreviation, the parameter matching forms mentioned in the previous couple of points will also match a term any of whose parameters, or its opid, matches the pattern.

You can delete the term at the point, i.e., replace it with a term-placeholder, with '(c-d)'[81], '(m-d)'[79], or '(c-a)(c-d)'[82].

The command '(c-d)'[81] is considered the "normal"; for detail, do a 'show'[55] or '(cm-(mouseright))'[312] on the relevant commands.

Insertion of a term X on top of a term Y will have an effect conditioned on the settings of three user settable bits:

```
left/right mode            '{(c-w)≡(m-w)≡(c-i)}'[12] toggles
insert/replace mode        '{(c-r)≡(m-r)}'[13] sets replace-mode
display-/term-struct mode  '(c-u){(c-r)≡(m-r)}'[14] sets term-struct mode
```

'(c-b)'[10] and term insertion commands restore these bits to normal, namely: left-mode, insert-mode, display-structure-mode

In insert-mode, the point Y will be replaced by X, then Y is put into the left-most(right-most) term-slot of X, if there is one.

In replace-mode '{(c-r)≡(m-r)}'[13], the point Y will be replaced by X.
The text slots of X will then be filled by the text children of Y from left-to-right (right-to-left). It is similar for the term slots of X and the term children of Y.
However, what is meant by children of Y and the left/right order is determined by the setting of the display-/term-structure mode '(c-u){(c-r)≡(m-r)}'[14].
N.B.: When a term is displayed, some of the text- or term-children of the underlying term may be elided from display, and the order of the children in display may differ from the order they have in the underlying term.

There are several possible sources for X, selected by the various edit commands.

the term clicked on	'(mouseleft)'[324] on <term to insert>+ and its variants
the term at the point (if not a slot)	'(m-(mouseright))'[321] '(c-c)'[94] '(c-a)(c-c)'[95]
top of stack (→doc_for_cut_n_paste {56})	'(c-p)'[87]
2nd from top (→doc_for_cut_n_paste {56})	'{(m-p)≡(cm-p)}'[88]
referenced operator instance	';'[4] immediately followed by object name ';;'[103] on an ABS or DISP reference (→doc_for_ob_refs {51})

Lots of commands are defined explicitly to insert specific operators.
The commands '(c-a)(c-?)'[28] and '(cm-(mousemiddle))'[311] will often show a keyboard command for inserting the operator that is at the point.

EXCEPTIONS: Replace-mode will usually be forced when inserting a 1-place text operator on top of an instance of a 1-place text operator. Thus, '(c-t)'[109] will usually convert a text operator to the appropriate form.
Pasting one inference steps is different;
VIEW→ doc_for_proof_refinement2 {27}.

doc_for_string_editing {20}

Terms have parts that are not themselves terms, rather they are strings, sometimes called operator-parameters since they are often conceived as parameterizing families of operators.

The "text-cursor", as opposed to the term cursor, usually must be located in the pertinent string part of the term in order to edit it. The point may be either on a term, or at a point in a string. When the mark is also in the same string, the substring that is between them is considered "selected" as the argument to certain edit-commands.

Setting the text cursor.

Moving the Point from Term to String(Text).
Assume the point is on a term:
When clicked on a string, ‘(m-(mouseleft))’[309] will place the point at the end of the string, and the mark at the beginning, no matter where in the string you click.
Once you have a text-cursor, you can move the point simply with ‘(m-(mouseleft))’[309].
It is often possible to move down into a string with ‘(c-j)’[62] so long as the ABS or display form for the term has the attribute "easyparm"; when this works, the point is left at the right end and the mark the left end, just as it is for ‘(m-(mouseleft))’[309]. In any case, you can navigate with the very simple commands ‘(m-j)’[56], ‘(m-k)’[57], ‘(m-h)’[59], and ‘(m-l)’[58], which will not affect the mark.

You can position the point to another text position by pointing at it with ‘(m-(mouseleft))’[309], which sets the point to the indicated character, and moves the mark to where the point was, thus selecting a substring if both are in the same text.

Within text, ‘(c-h)’[65] and ‘(c-l)’[66] move one character left or right, and ‘(c-a)(c-h)’[67] and ‘(c-a)(c-l)’[68] move the point to the first or last position in the string. ‘(c-j)’[62] will also move to the front of the string containing the point. So, with "easyparm" operators, repeating ‘(c-j)’[62] will move the point successively from the term to the last string position, to the first.

Insertion and deletion. VIEW→ doc_for_string_editing2 {21}

(Continued from VIEW→ doc_for_string_editing {20}) doc_for_string_editing2 {21}

Insertion and Deletion in Text.

‘(c-a)(c-d)’[82] deletes the whole string and clears any "meta-bits".

‘(c-d)’[81] inserts the null string into an empty slot, and otherwise, deletes the selected substring, between mark and point, and replaces the top of the term stack with it, in a term built from the operator [text]. It often happens that text-commands adjust the point and mark to select a single character in order to avoid mishaps. Examples of such commands are ‘(mouseleft)’[324] and the various left- and right-motion commands mentioned above, and the ‘(c-d)’[81] command itself.

‘(m-d)’[79] deletes the selected text.

(rubout) deletes the character before the point.

Ordinary keboard input characters, not control or meta, are self inserting before the point.

‘(c-y)’[84] replaces the top of the term stack with a copy of the selected substring.

‘(c-p)’[87] copies the top of the term stack (if it’s text) to before the point.

‘{(m-p)≡(cm-p)}’[88] copies the second-from-the-top of the term stack (if it’s text) to before the point.

‘(c-()’[89] replaces the top TWO elements of the term stack by copies of the selected text.

‘(c-))’[90] pops the term stack.

To insert special non-keyboard characters from the Nuprl Font into a string, look in the object VIEW→ words_menu {63}, which can be raised with the WORDS {63} button from the VIEW→ main_menu {2}; you will find the special characters listed there in several little bunches, and ‘(mouseleft)’[324] on such a character there will copy it to before the point.

‘(c-$)’[126] will set the abstraction meta-bit, which is used in ABS objects.

‘(c-<)’[127] will set the display meta-bit, which is used in DISP objects.

‘{(c-n)≡(m-n)}{(c-$)≡(c-<)}’[128] will clear any meta-bits without changing the string.

Often edit commands are overloaded depending on the display-forms or operators the point is located on or in, and perhaps on the kind of object in which the command is being executed.

There are 3 major contexts used by this editor, as well as some lesser ones that are handled less systematically.

1. The PLAIN context is really just the lack of any special context: it is the one in which one performs most editing of mathematical notation, for example.
2. The ML context is used to interpret commands for editing structured ML expressions. VIEW→ doc_for_ML_editing {24}
3. The WORDS context is used to interpret commands for a simple word processor, which is used not only for creating text, such as you are reading here, but also provides the generic "glue" for informally combining various kinds of expressions together into a single term, by providing some white-space and margin control for example. VIEW→ doc_for_WORDS_editing {29}

Some minor contexts supported by this editor are: operator definitions, display-form specifications, operator-precedence descriptions, and inference rule definitions.

When one wants to mix different kinds of expressions, there will be moments in the editing process, namely, when one is trying to embed one kind of expression in a context for another kind, when the command you want to give cannot be used in the context. For this reason, each major context has an associated "temporary environment" which can be inserted in order to give the appropriate interpretation to the otherwise ambiguous command.
These temporary environments and the commands for inserting them are:

‘ml’[146]   ML→ <ml>
‘wo’[292]   WORDS→ <words>
‘{pl≡plain}’[76] PLAIN→ <term>

If point-suppression (toggled by ‘(c-u)(c-s)’[33]) is enabled as is normal, these environments will be removed if the point is located onto them, and the special ones, i.e. not PLAIN→ <term>, will even be removed if the point is just below, so long as the operator is native to the context.
For example, to insert <ml> <ml> in a WORDS context, issue the commands ‘ml’[146] ‘(c-(space))’[104]. If you had just entered ‘(c-(space))’[104] in the WORDS context you would have gotten <words> <words> instead.

The prompts for term slots in special contexts are usually indicative of the the appropriate context. For example, ordinary juxtaposition of words, and function application in ML are usually indicated simply by separating the two components by a space or line break.
Guess which is which: <ml> <ml> versus <words> <words>

doc_for_comment_insertion {23}

Comments may be inserted in most terms, and they will be ignored when the system uses a term for special purposes.

A comment is a "cover" and so can be removed from the term it adorns with the usual delete command ‘(c-d)’[81].

The command ‘com’[270] inserts a comment cover on the point. If the direction mode is left-to-right (normal) then

```
  {[word] } <term>
```

is inserted. If the mode is right-to-left, then

```
  <term> {[word] }
```

is inserted.
The command ‘(backspace)’[99] will toggle between these forms.

EXCEPTIONS: In this version of Nuprl, comments may not be used in the right-hand-side (definiens) of an ABS object, nor in the model of a display form spec in a DISP object.

The primary form of ML in this editor is "structured". doc_for_ML_editing {24}

(For unstructured ML and the underlying scanner, VIEW→ doc_for_ML_scanner {72}.)

Most of the ML syntax has been represented with Nuprl operators.

Do you want to see a listing the basic ones? YES

(As usual, ‘(c-a)(c-?)’[28] or ‘(cm-(mousemiddle))’[311] on an operator will tell you something about it, especially if there is a command for inserting it.)

There is a utility menu for editing ML which you can raise either with the ML {62} button in the MainMenu {2}, or with the ‘mledit’[276] command, or with the ‘{ed≡edit}’[271] command in an ML context.

There are a few special methods involved in editing structured ML.

VIEW→ doc_for_ML_editing_let {73} VIEW→ doc_for_ML_editing_if {74}

In-place evaluation using ‘{(m-z)≡(cm-z)}’[291] is key; HERE for details.

Name completion on ML global ids is quite helpful.
‘(c-&)’[23] ‘(c-a)(c-&)’[24] ‘(c-u)(c-&)’[25] VIEW→ doc_for_name_completion {41}

‘show’[55] and ‘(cm-(mouseright))’[312] search the ML objects in the library for definitions of ML global ids. HERE for detail.

ML ids are represented as terms with operator [text]. Usually, entering an identifier on a term slot will insert the [text] using that identifier as the text. But if that doesn’t work, say because the identifer is a keyboard command, then you may have to insert the identifier with the ‘(c-t)’[109] command.

To insert non-ML terms into ML code, you may need to insert a temporary context of an appropriate sort. VIEW→ doc_for_edit_context {22}

Please, review the commands for ML editing; most of them are simple insertion commands.

DO→ Review cmds for group: "ML"

If your ML editing commands don’t seem to be working correctly, especially if you are getting non-ML operators, then you are probably not in an ML editing context; use the ‘ml’[146] command to temporarily insert an ML context (not to be confused with the ‘ml(return)’[148] command).

How to find documentation about ML: VIEW→ doc_for_ml_doc {64}

For explanation of the term representation of proofs, see ProofTerms. {67}

Proof terms may be as freely edited as any other terms, so you may cut-and-paste them together or build them explicitly without regard for validity. But of course, we normally use specific utilities to help us form proof terms that consist of actually legitimate inferences by executing a tactic at each step.

THM objects.

THM objects may contain only proofs in which every step is legitimate according to the basic rules of inference. See PostingTHMs {75} for how to post theorems to the library, and so making them available for further use as lemmas.

Starting a proof.

If you initiate a proof by creating a new THM object, say with the NEW button in MainMenu {2}, or view an existing THM object, You will be given a proof term which you may then modify.

If you want to build a proof from scratch, unmediated by a THM object, simply create a "mathematical" expression (i.e., not a WORDS {29} document, nor ML Code {24}, nor other special purpose system expression), then use the generic execution command '(c-z)'[290].

This will create a sequent with no hypotheses, perhaps inserting some universal quantifiers for variables you left free. You should edit this term until no free variables are left, for which purpose you will find the generic adjustment command 'adj'[125] useful.

CONTINUE {26}

Making Proof Steps. (For proof term structure VIEW→ doc_for_pfterms {67}.
For starting a proof VIEW→ doc_for_proof_editing {25}.)

‘(c-z)’[290]

The most common procedure for building a proof step is to fill in the tactic-code part of an inference step form with ML Code {24} then issuing the generic execution command ‘(c-z)’[290]. ‘(c-z)’[290] will jump up to the smallest proof step containing the highlighted point, and "refine" the goal to produce subgoals.

Note: Whatever subproofs may already be there will be replaced by the new subgoals, if any. Non-trivial subproofs that are discarded will be popped up in another window so you get another crack at using them, say for cut-n-paste.
If you know you don’t want these subproofs, delete them before using ‘(c-z)’[290]. To delete them, locate the point on the group of subgoals as a group then use ‘(c-d)’[81]; you don’t need to move the point back to use ‘(c-z)’[290]. The ‘(mousemiddle)’[313] command is most useful for moving the point to a large term -- play with it.

The Auto Tactic.

Many simple inferences are knocked off by the tactic Auto. It is so common to try it, that the ‘(c-z)’[290] command has been tailored to insert it automatically in some circumstances. If there are no subproofs, and the tactic code slot is empty, then ‘(c-z)’[290] will insert Auto and execute.
There are three common variants of the auto tactic: Auto Auto’ SIAuto
‘(backspace)’[99] will cycle among them. (No explanation given at this point.)

Subgoals are often labeled "wf" for well-formedness goals, which assert that expressions have certain types. Often Auto will be useful on these subgoals even when it may be worse than useless on other sugoals. When this is the case, we don’t want use (T ....), but instead we want a variant (T ...w) that applies Auto after T only to the well-formedness subgoals. The command ‘{au≡auto}w’[208] will append Auto this way.

See {94} EXAMPLE {94} for an exercise using these methods.

VIEW→ doc_for_proof_refinement2 {27}

(continued from VIEW→ doc_for_proof_refinement {26})

More on Making Proof Steps. (See also PF and TacMenu1 {28} for more handy devices.)

Mini-menu for tactics.

Besides Auto the most common tactics can be invoked with '(mouseleft)'[324] clicked on the turnstile or hyp number within a goal. This will insert a little menu at the point of tactics pertinent to that hyp or conclusion. Actually, if the point is an empty slot, usually within the tactic code part of a proof, the clause number will be inserted, and you must click again on the same hyp or turnstile to get the menu inserted at the point.

Kreitzing. 'kr'[259] '{(c-n)≡(m-n)}kr'[262]

It often happens that after you have built several inference steps, you find them too fine-grained and wish that all the steps were compacted into one. If you locate the point on the subproof you wand to treat as a single step, the command 'kr'[259] will compose all the tactic code for the subproof into a single tactic and re-execute.

This conversion is called "Kreitzing" after Christoph Kreitz, who was the first to employ such compression systematically.

There is also a heuristic for undoing kreitzed code, which is invoked by '{(c-n)≡(m-n)}kr'[262].

Rerunning Proofs. 'run'[265]

The subgoals of a legitimate Nuprl proof are always determined by the goal and the tactic code. This means that a proof can be completely recreated from the top goal and the tree of tactic code gleaned from the proof tree. If one takes a proof and modifies it, say by changing the goal or perhaps some of the internal tactics, rather tan rebuilding the analogous proof step-by-step, one can simply rerun the proof from the goal down, regenerating the subproofs.

The 'run'[265] command will do this. If for some reason the proof cannot be run below a certain point in the proof, say because some tactic fails or the number of subgoals is not as expected, an error message is inserted at that point and no further progress down THAT subproof is attempted.

The user may then attempt to make repairs at that point or perhaps higher, then rerun the proof from the corrected point down.

Copying one inference on top of another.

When an inference step term, B, is copied on top of another, A, say with the '(m-(mouseright))'[321] command, the goal and address of A are kept, but the tactic code and subproofs are taken from B. If the goals are different, then the new step will be rerun as with 'run'[265] above.

Menu2 PF tactic_menu1 {28}

ChainRelation: <hungry>$^+$
ApFun: <fun (hungry)> to: <arg (hungry)>$^+$
CoerceToType ⌈<term (hungry)>⌉ <ml>$^+$
TypeTagging[num]: <hungry>$^+$

At Type{[level]} <ml>$^+$

Repeat <ml>$^+$
[num] Times <ml>$^+$

Analyze'[clause]$^+$
Witness[clause]: <terms (hungry)>$^+$
Select[clause]: 1 |2$^+$
FunExtensionality$^+$

<terms (hungry)> Asserted$^+$ Enough: <terms (hungry)>$^+$
<terms (hungry)> By <ml>$^+$ Decide: <terms (hungry)>$^+$
DescribeFun: <var> by <∀∃!>$^+$

SimilarTo: <hungry>$^+$
BackThru: <hungry>$^+$
FwdThru: <hungry> on [<terms (hungry)>]$^+$
Inst: <hungry>$^+$
<ml> Using:{<levels (hungry)>}[<terms (hungry)>]$^-$

Rewrite[clause] by <hungry>$^+$
Subst[clause]: <hungry>$^+$

New:<vars> <ml>$^-$
Rename[num]: <var>$^+$
CBV[num]: <term>$^+$

<ml> {Auto'}$^+$ <ml> {Id}$^+$
<ml> {Try^{0}Complete Auto}$^+$
<ml> {Auto sans <pattern (hungry)>}$^+$
<ml> $^-$<hyp>$^+$

⌈<term (hungry)>⌉$^+$
Tms:[<terms (hungry)>]$^+$
subst:{<levels>}[<terms>]$^+$
RuleArgs:[<terms>]$^+$

doc_for_WORDS_editing {29}

The WORDS operators are formally meaningless devices for representing words, punctuation, and collections of terms with control of spacing for display. One often embeds other terms in WORDS to form documents or menus.

[word] represents a single word usually. If you enter a string with spaces, it will usually be automatically burst into a sequence of words. You can insert it with ‘(c-t)’[109] or just by entering text at an empty term slot.
TIP: It is often easier to restructure your WORDS if punctuation is represented as the last character of the word preceding it.

For WORDS pairing operators VIEW→ doc_for_WORDS_pairing {30}

[left paren]<words>[right paren]
is a simple parenthesizing operator. There are specific commands for inserting various specific versions, such as ‘{\)≡\(}’[297] to insert (<words>), and ‘"’[294] to insert "<words>".

[chars]<|←words> sets a left margin for the subterm. The character string will precede the subterm to the left of the new margin; This string is almost always zero or more spaces, and usually zero. Insert with ‘margin’[296]; ‘(c-u){(tab)≡(return)}’[102] will put the null string in the string slot, so ‘margin’[296] ‘(c-u){(tab)≡(return)}’[102] will simply set the local margin.

For the edit commands specialized for WORDS to work, you need to be in a WORDS edit context.
If you are not in a WORDS context, then you may temporarily insert one (namely WORDS→ <words>) with the ‘wo’[292] command.
You may also need to insert another kind of context when you want to use non-WORDS commands inside a WORDS document.
VIEW→ doc_for_edit_context {22}

Layout of WORDS is unnecessarily slow, and we plan to improve it later.

DO→ Review cmds for group: "Words"

WORDS pairing operators. (continued from →doc_for_WORDS_editing {29})

<words> <words> is the basic WORDS sequencer, which breaks when there's no more space. '(c-(space))'[104] inserts it, and it is often created by an automatic burst of a word with spaces. You can also get it from any of the forced-break forms below with '(c-a)(c-(backspace))'[101].

<words><words> jams the two terms together with no break possible. You can get it from the previous form by '(backspace)'[99].

<words><words> jams the two terms together but a break is possible. You can get it from the previous form by '(backspace)'[99], and '(m-(space))'[105] inserts it.

<words>: <words> "labels" the second term with the first. The second term has its own left margin. You can get it from the previous form by '(backspace)'[99].

<words>
<words> forces a break between the two subterms. '(m-(return))'[107] inserts it, and you can get it from any of the previous WORDS pair forms with '(c-a)(c-(backspace))'[101].

<words>

<words> forces a double break between the two subterms. '(cm-(return))'[108] inserts it, and you can get it from the previous form by '(backspace)'[99].

You will sometimes need to reorganize these sequences of terms.
VIEW→ doc_for_term_sequences {31}

In the Nuprl system, for many purposes a term sequence will be represented by a term, using a binary operator as the catenator.

Let OP be an opid and a list of op-parameter values. Then we say a term T is an OP-representation of a certain term sequence as follows:

1. If T is an instance of an operator with OP as its opid and op-parm-values, and has no immediate subterms, then T represents the empty sequence.
2. If T is an instance of an operator with OP as its opid and op-parm-values, and has two-immediate subterms and no bindings, then T represents the catenation of the sequences represented by its subterms.
3. Otherwise, T represents the unit sequence with itself as the only member.

The system avoids ambiguity by expecting specific operator ids and op-parms to be used in specific places where term sequences are needed as data. The structure of the particular binary trees representing the sequences will be ignored for those purposes.

It often happens that a person wants to reorganize a term representing a sequence so that it represents the same sequence, but representing a certain segment with a single subterm.
To select a subsequence,
1. click ‘(m-(mousemiddle))’[315] anywhere inside one term you want at the beginning or end, which will insert #MARK <term>,
2. then with this mark left highlighted, use the same command to click on the other endpoint; if you move the point off the #MARK <term> when you do the second ‘(m-(mousemiddle))’[315], you’ll just get another #MARK <term>.

2’. If you use ‘(c-a)(m-(mousemiddle))’[316] instead, regrouping will work even through operators that merely have the same opid and sequence of parameter KINDS. This is useful when regrouping in WORDS contexts.

The smallest sequence representation containing these two subterms will be reorganized to get these elements into a single subterm presenting the segment with them as endpoints.

‘norm’[140]
You can also reorganize a sequence so that nesting is only through one place. The command ‘norm’[140] will perform this nesting in the direction indicated by the left-/right-mode, toggled by ‘{(c-w)≡(m-w)≡(c-i)}’[12]. But unlike most of edit commands, it uses the positioning according to the term structure rather than the display structure.

Operator definitions are given in ABS objects of the library.
They look like this:

```
<conditions>
<ab lhs> == <ab rhs>
```

or like this, when there are no conditions:

```
<ab lhs> == <ab rhs>
```

Recursive definitions are permitted (they are automatically translated to non-recursive form using the Y-combinator).

The operator definition does not take effect until a successful "verification" has been performed, typically with the '(c-v)'[284] command which will also checks for coherency and raises a utility for developing auxilliary objects such as display forms and typing-theorems. VIEW→ doc_for_abs_verify {76}

The "conditions" are used by the system and the editor for various purposes, but for normal mathematical operators, there will usually be no conditions attached to the definition. VIEW→ doc_for_term_sequences {31}

If you need to reveal the hidden conditions slot in order to add some conditions, just use '(c-s)'[31] on the defintion form. '(c-t)'[109] and identifers will insert a condition in a condition slot; '(c-(space))'[104] will insert <conds>, <conds>

If the right-hand-side of the definition is PRIMITIVE, then the ABS object merely declares the operator without defining it to expand to anything.

There is a small utility menu for editing ABS objects, which you can raise with the 'absedit'[273] command or, if you are focussed on an ABS window, with the '{ed≡edit}'[271] command.

The easiest way to create an operator definition is to fill in the right-hand-side with the definiens, then use the 'adj'[125] command, which will adjust the left-hand-side so that it makes sense with respect to the right hand side; and when the definition is recursive, the rhs may be adjusted as well. HERE {33} for detail on adjusting.
Even if you just want to declare a primitive operator, it is usually easiest to make up a bogus rhs to force the lhs, and then replace the right-hand-side by PRIMITIVE.

Inserting new operators without defining or declaring them: It is also possible to insert <ab lhs> == <ab rhs> directly anywhere with the 'abs='[121] command. This is the recommended procedure for inserting an operator which has no definition and no available examples. You create the term indirectly using 'adj'[125] on the "definition" until the left-side is the term you want, then replace the abs-equality by the left-hand-side, say with the '(c-u)(c-(space))'[78] command applied to the left-hand-side.

DO→ Review cmds for group: "Abs"

doc_for_edit_adjust_op_form_by {33}

Adjust lhs to make it a reasonable left-hand-side of an operator definition with right-hand-side rhs.

1. If the lhs is a var, its name will become the opid.
2. If the lhs is blank, the (current_object ()) will become the opid.
3. Edit slots will be eliminated from the lhs.
4. Free vars and level expressions will be imported from the rhs if they aren't already in the lhs. New level exprs are added to the front, new parms and new term vars are added to the back.
5. The arities of free vars from the rhs supercede the arities of free vars on the left with the same name.
6. The bindings of the lhs are adjusted to match the arities of the var instances in their scope.

The rhs will also be adjusted when it seems that a recursive definition is desired.

7. If the lhs is blank, then blank
   terms on the rhs will be filled in with the new lhs.
8. If the rhs contains instances of the lhs operator, they will be updated to instances of the new lhs.

doc_for_display_form_edit {34}

Display form specification can be rather complicated, so we will not give a complete explanation here. More documentation on display methods will be added in time. For more complicated methods of display, it is suggested that examples using those methods be found and emulated. The ‘vdf’[45] command will raise the DISP object being used to display the point.

The use of the utility window is more likely to be necessary than for other editing-domains in this editor.
That utility can be raised with the DISP {77} button of the MainMenu {2}, or with the ‘dfedit’[272] command, or from inside a DISP object with the ‘{ed≡edit}’[271] command.
All the basic devices and operators you need to define display forms are available via this utility.

A DISP object must unconditionally expand (use of operators as macros in DISP objects is very common) to a sequence of display-form specifications. The sequencing operator is

```
is   <specs>
     <specs>
```

and an individual specification has this form:

```
 Model: <model>  Attrs: <attributes> |  <Formats>
 -----------------------
```

When the system tries to display an operator instance, it looks for the last display spec whose model matches the operator instance. (There may also be several DISP objects in the library that apply to the operator.)

It is also possible for the user to temporarily disqualify, or suppress, a display form from being used, either at the point, in a window, or globally, in order to use a temporarily more desirable one. VIEW→ doc_for_display_suppression {16}

Indeed, "Model: <model> | fu
_______________________" is normally displayed in a way that hides the attribute- and format-fields unless the point is on the spec or in one of those fields, or one of them has empty slots. To reveal these hidden parts throughout the window, use ‘(m-s)’[34] on the spec.

Quick Display Form Creation: VIEW→ doc_for_display_form_imitation {35}

Display Form Models: VIEW→ doc_for_display_form_models {36}

Display Form Formats: VIEW→ doc_for_display_form_formats {37}

Display Form Attributes: VIEW→ doc_for_display_form_attributes {40} doc_for_display_form_imitation {35}

It is often easiest to imitate an example, rather than create from scratch. In the case of defining display forms, this is facilitated by the ‘adj’[125] command used in a DISP object.

Just insert an instance of an operator whose display you want to imitate, and use the ‘adj’[125] command. All the display forms for that sample will be inserted instead into the DISP object, with the new opid and permuting the op params and arities if necessary.

To use this, it is necessary that the name of the new DISP object is gotten from the name of an active ABS object by adding _df, and that the op parameter types and the arities of the sample and the ABS operator match, modulo permutation.

Usually, one will want to make some alteration to the display method appropriate to the new operator. VIEW→ doc_for_display_form_edit {34}

In

```
 Model: <model>  Attrs: <attributes> |  <Formats>
 -----------------------
```

the model is a term which is a example of the operator to be displayed. The terms whose display is being specified by the display-form spec are the instances of the model gotten by filling in various parts of the model; so the model serves as a kind of schema. There are two kinds of such schematic "variables" pertinent to display, namely, display templates for terms, which look like <[template]>, and display-meta-variables for strings and binding variables belonging to the operator, which are indicated by putting angle brackets around the string (but these angle brackets are not actually part of the string-value itself).

For example, if we wanted to make a simple display form for the lambda operator λ[var].<*>, we could use display spec whose model was λ<x>.<t>.

A display-meta-variable for text in a operator or binding variable may occur more than once, in which case, for a term to match the model it must also have the same strings filling those positions.

If a display-meta-variable with the null text is used in a binding position of the model, then a term will match only if the binding variable matching that position doesn't really bind anything in the instance. This way we can give different methods of display that distinguish this failure to bind.

(More kinds of model matching are significant. Not documented here yet. See Paul Jackson's manual. VIEW→ doc_for_using_pauls_manual {8})

The 'adj'[125] command.
The model can usually be created automatically by using the 'adj'[125] command. This works when either the model position is empty and the DISP object name is a "_df" extension of a verified ABS object, or else the model position is filled by an obref to a verified ABS object.

Remember: you don't have to fill all the places of the model with meta variables; you may use ordinary strings in the text-slots, and ordinary terms in the term slots, to require an exact match in those places.

HERE for review of display model edit commands.

VIEW→ doc_for_display_form_edit {34}

Display Form Formats. doc_for_display_form_formats {37}

In

```
 Model: <model>  Attrs: <attributes> |  <Formats>
 -----------------------
```

the Formats section should be filled by a term constructed from various special operators that explain how to display an instance of the model.
VIEW→ doc_for_display_form_models {36}

[text] print the text-string. (Insert with '(c-t)'[109].)

<[?]:[prompt]:<child attributes>> displays a term or text-string matching a display schema variable in the model.
VIEW→ doc_for_display_form_dfchild {38}

<Formats><Formats> do what the first part says, then the second. (Insert with '(c-(space))'[104].) VIEW→ doc_for_term_sequences {31}

<disp>←LINEAR declares "linear" mode for all the enclosed format commands, which prohibits breaking. Elision may result. (Insert with 'linear'[257].)

{[hard]}
causes a break to new line to occur unless we are in "linear" display mode at this point in display. The enclosed text-string, spaces usually, is printed if there is no break. (Insert with 'hard'[258].)

<disp>
←MARGIN sets the left margin for any breaks caused by breaking commands it governs, i.e., breaking commands it encloses, but which are not enclosed by an innermore occurrence. (Insert with '{margin≡indent}'[255].)

[[break]]
may cause a break, depending on whether it's needed to avoid elision, and on what <disp>←CONSIS govern it. (Insert with 'break'[254].)

<disp>←CONSIS is used to indicate that the [[break]] format commands it governs my break if need be to avoid elision, but that all those it governs will break if any break is produce by any of the format commands it encloses, i.e., it indicates "consistent" breaking. (Insert with 'soft'[256].)

VIEW→ doc_for_display_form_edit {34}

A display child format command has the form

<[?]:[prompt]:<child attributes>>

which has two text slots and a term slot.

Let's consider an instance of this operator in which the attributes-slot is filled by "*", indicating no parenthesizing. When this slot is filled by "*", that part is not shown, so we can have an instance looking like this: <x:int>.

For example, here's what the format command <x:int> means:

Display the string or term matching the display-scheme metavariable named "x" in the model.

But if the thing matching the meta-variable is an empty slot, display the string "int" instead, as a prompt.

Do not generate parentheses around the display of this child.

The slot attributes are mostly used for control of parenthesization, which is explained in VIEW→ doc_for_parentheses {95}.

To read about creating and editing display child format commands, VIEW→ doc_for_display_form_dfchild_editing {39}.

VIEW→ doc_for_display_form_formats {37}

doc_for_display_form_dfchild_editing {39}

Here we will discuss how to enter a display child format command.

In VIEW→ dfedit_big_menu {77} you will find a button ChildMenu which, when applied to a term, will generate a menu of insertion buttons for all the display schema variables in the term. For example, applying it to λ<x>.<b> will generate

```
#COVER
# <x:[prompt]>+
  <b:[prompt]: * |Other>+
#
λ<x>.<b>
```

You can then use these insert buttons to copy the child format commands into the format part of the display spec. When you're finished, just locate the point on the "cover" and delete with '(c-d)'[81].

This little menu is also generated if you use the 'adj'[125] command on an obref to create the model.

In VIEW→ dfedit_big_menu {77} you will also find a button DFchild which will insert

<[?]:[prompt]: * |Other>.

You may then simply enter the strings, then select from the choices given for parenthesis control. (This is a generic method for offering choices, described in VIEW→ doc_for_edit_choice {78}.) Selecting Other will give you some more choices; see VIEW→ doc_for_parentheses {95} for how to decide what you want.
If you want to change this slot, you can reinsert the choices with the * |Other+ button in DisplayForm EditMenu {77}.
You can also use '(backspace)'[99] to cycle through * (≤self) (<self).

You can also insert <[?]:[prompt]: * |Other> in the formats area with the '(c-<)'[127] command

VIEW→ doc_for_display_form_dfchild {38}

Display Form "Attributes". doc_for_display_form_attributes {40}

In

```
 Model: <model>  Attrs: <attributes> |  <Formats>
 -----------------------
```

the attributes section is filled in with a sequence of terms the system interprets in various ways. You will find buttons for inserting these in VIEW→ dfedit_big_menu {77}. If the attributes field is hidden, try ‘(c-s)’[31] on the whole display form spec.

Open form makes it possible for instances of this display form to be parenthesized when occurring in suitable positions in other terms. I.e., if this attribute is missing, then the display form is considered "closed", no disambiguation needed. VIEW→ doc_for_parentheses {95}
N.B. whether *Open form* is among the attributes of a spec is toggled by ‘(backspace)’[99] on the spec itself.

InsertCmd:[name] declares a name that can be used instead of the object name when inserting the display form using the ‘;’[4] command-prefix.
N.B. whether InsertCmd:[name] is among the attributes of a spec is toggled by ‘(c-a)(c-(backspace))’[101] on the spec itself.

[condition] is used to condition macro behavior on the operators. It’s the same as is used in ABS conditions. The CondsMENU button in DisplayForm EditMenu {77} will show you what a variety of selected conditions are used for.

NotAtPoint indicates that this display form should not be used when the point is located on the term. This feature is toggled throughout a window with the ‘(c-u)(c-s)’[33] command.

<attrs>; <attrs> is the sequencing operator, with NOattrs being the empty sequence. VIEW→ doc_for_term_sequences {31}

To find out what the other attributes in DisplayForm EditMenu {77} mean, you may use the Buttons button on the insert-buttons.

VIEW→ doc_for_display_form_edit {34}

‘(c-&)’[23] ‘(c-a)(c-&)’[24] ‘(c-u)(c-&)’[25]   doc_for_name_completion {41}

Various kinds of string-representing operators have an attached name-completion utility invoked with these macros.

‘(c-&)’[23]	basically looks for right-end-extensions of the string, and is usually much faster than full name-completion. It’s actually a little more complex than prefix matching: if the string we’re trying to complete contains no upper case letters or ‘_’ characters, then matches are considered any strings whose lower-casing and ‘_’-deletion has the pattern string as prefix.
‘(c-a)(c-&)’[24]	is like ‘(c-&)’[23], except that it uses contiguous-substring instead of prefix as a match.
‘(c-u)(c-&)’[25]	If there is a cache associated with the utility for the relevant kind of string-representing term, this will refresh it.

In every form of name completion, if there is no match, the sample is deleted (but saved on the stack).
If there is one match, it replaces the sample.
If there are a few matches, an option sequence replaces the sample.
VIEW→ doc_for_edit_choice {78}
If there are lots of matches, a separate window of insertion buttons is created as scratch; they are still presented as an option sequence to facilitate narrowing the list.
If too many match, the user is warned and given an option to list them anyway.

Currently, here are the kinds of name completion implemented.

ML name completion, which searches a cache of global ML names, is invoked on instances of [text].

Object name completion, which searches all object names of the library, is invoked on instances of:
   VIEW→ [ob name] and it variants,
   ‘[mltok]‘, since often tokens are used in ML to refer to lib objects,
   and other miscellaneous forms usually referring to objects:
    =[obj].[name] ob:[ob name] *[object]* [ob].[name]

Primitive edit cmd name completion is invoked on instances of *[primitive cmd name]*.

Internal defined edit cmd name completion is invoked on instances of [[internal macro name]].

File Name completion, which is limited to ‘(c-&)’[23] and is invoked on WORDS constructs and ML string literals of form "[mlstring]".

In order to simplify transitions between closely related terms, there is a generic extensible toggle facility. These "toggles" typically cycle through several variations on a form of term.

There are two generic toggling commands, ‘(backspace)’[99] and ‘(c-a)(c-(backspace))’[101], called the "toggle" and the "long toggle". The long toggle is meant to connote a greater dissimilarity between the related values.

When one of these toggle commands is given, a list of ML functions is successively called until one succeeds; it’s a no-op if they all fail. The arguments passed are the term, and perhaps the left/right bit, and an indication of which of the two toggle commands was used.

The command ‘(c-a)(c-?)’[28] will indicate whether a toggle command will have an effect on the point.

To see the descriptions of the toggle routines, click HERE.

How to search for terms in a window. doc_for_term_search {43}

‘find’[20] ‘findop’[21] ‘(c-a)findop’[22] ‘findsame’[19] ‘(c-#)’[18]

If, in a window, you apply the ‘find’[20] command to a proper subterm, say PAT, of the window, the whole window will be searched for subterms that match against PAT according to the document VIEW→ doc_for_edit_match {18}.
Ordinary terms simply match themselves.

‘findsame’[19] will search the point term using the previous pattern. It’s mainly used for searching in a different window than where the pattern was set.

If you apply ‘find’[20] to the whole term of the window, then no search is done, but the whole term is set as the search pattern for subsequent use of ‘findsame’[19]. This is useful for creating a search pattern in a separate window.

‘findop’[21] searches for terms matching the operator at the point, actually matching ‘(c-o)’[85] of the point.

‘(c-a)findop’[22] is like ‘findop’[21] except it matches ‘(c-a)(c-o)’[86] of the point term.

If you simply enter an identifier string preceded by "’ {44}", then a search is done for terms containing the string as a substring of the opid or a parameter.

Matching subterms are wrapped in the cover <words>, which is easily removed with ‘(c-d)’[81]. A preorder seach for the next <words> is caused by ‘(c-#)’[18].
And if there are more than one matching terms, then the whole window will be wrapped in

```
CLEARmarks
# Found. (Use ‘(c-#)’[18] or NextMark to advance.)

<words>
```

which contains some utility buttons. All the <words> marks will be removed if you use ‘(mouseleft)’[324] or ‘(c-z)’[290] on the whole.

’ edit_ident_search_char {44}

How to create new objects and operator definitions.   doc_for_object_creation {45}

Use the NEW button in MainMenu {2} to open an object creation utility. (Try it now.)

This gives you some choices. As usual, if you don't select an option from
 first-option |second-option |more-options, then the first option is used.

The "newob" option creates and views a new object of the specified kind at the indicated place.
The "newsegment" option creates some objects for managing a contiguous section of new objects.

A place is indicated as follows:

  LAST means put it AFTER ALL other objects.

  (After: →[ob name]) means put it AFTER the named object.

  (Before: →[ob name]) means put it BEFORE the named object.

  DEFAULT means to use a default previously specified by the user.

When "newsegment" is used, the DEFAULT is for new objects to be inserted at the end of the new library segment. The DEFAULT can also be set via a button in the object creation menu that NEW raised.

DEFINING new operators along with their display forms and well-formedness theorems:

  Operator definitions or declarations are made in ABS objects. So create an ABSobject with whatever name you like, normally mnemonic for the operator, edit the definition and verify the object with '(c-v)'[284].
  This will raise a utility prompting you for creation of display forms and usually well-formedness lemmas specifying the operator type.

  See the following for how to create and edit definitions, display form specifications, and well-formedness lemmas.

  Edit Operator Definitions (ABS) {32}
  Edit Display Form Specs {34}
  Edit Proofs {25}

doc_for_ml_execution_cmds {46}

The main ways to invoke ML on user expressions are to use the commands '(c-z)'[290] and '{(m-z)≡(cm-z)}'[291], and to use the '(mouseleft)'[324] on certain buttons.

'(c-z)'[290] VIEW→ doc_for_ml_execution_generic {47}

'{(m-z)≡(cm-z)}'[291] VIEW→ doc_for_ml_execution_in_place {48}

'(mouseleft)'[324] on DO→ <ml>, FN→<ml> and its variants.

VIEW→ doc_for_ml_execution_mouse {49}

To abort execution, enter (c-c), or perhaps (c-c)(c-c), into the LISP window; then you can either continue execution with ":c", or abort with ":a" and resume nuprl with "(nuprl)".

You can exit the editor and key strings into the ML command interpreter in the LISP window by using the command (m-x)cmd. To get back to the editor from there, assuming there's at least one object on view, tell ML this: x(return)

---

doc_for_ml_execution_generic {47}

Executing ML code with '(c-z)'[290]. VIEW→ doc_for_ml_execution_cmds {46}

If the point is within a term whose operator is marked with the ml_string condition, such as:

DO→ <ml>

DO→ <ml>

ML> <ml command>
<term>

APPLY> <ml fn>
<term>

jump to the smallest one containing the point, else stay where you are, then:

1. If it's ML> <ml command>
   <term>
   then execute the command leaving the answer in a certain space provided for it.

2. If it's APPLY> <ml fn>
   <term>
   then replace the covered term by the result of applying the ML function to it.

3. Else, if you're in a Goal or Rule box for a THM, then check it with the possible effect on the subproofs.

4. Else, evaluate the term in ML and leave the textual answer on the stack.

It is easy to open up a command area wherever you are by inserting the cover
ML> <ml command>
<term>
with 'ml(return)'[148] and it easy to remove the cover with '(c-d)'[81].

doc_for_ml_execution_in_place {48}

Executing ML code with ‘{(m-z)≡(cm-z)}’[291]. VIEW→ doc_for_ml_execution_cmds {46}

Execute the term in ML, and return the textual response, indicating the type and value, or an error message, in a cover.
For example, you can easily find the type of an identifier this way; using this command on the identifier quicksort gives,

```
NOTICE: - : ((* -> * -> bool) -> * list -> * list)
quicksort
```

If the ML name cache is active, which is the default (CLICK HERE {91} if you need to activate it), and the term is more than just an identifier, then the lambda closure of the term (using its free identifiers that aren't cached) will be evaluated instead. This is especially handy for finding the types of subexpressions, and the types expected for its free identifiers.
Using this command on (last (hd a.b) 1) produces

```
NOTICE: a,b  - : (((int -> *) list # (int -> *) list list) -> *)
(last (hd a.b) 1)
```

The cover (NOTICE: <term>
<term>) is easily removed with ‘(c-d)’[81] or ‘(mouseleft)’[324].

DEFICIENCY: The utility misbehaves when evaluating a subterm that happens to contain cached ML identifiers that are bound above the point. A version of this function that would scan up for the context was too slow, so this inferior functionality was adopted. When you have no interest in the global binding of some identifier that you're using locally, it is sometimes worthwhile to remove the identifier from the cache with the aptly named button in the ML edit menu (ML {62}) which you should be able to raise through the main menu (VIEW→ main_menu {2}).

doc_for_ml_execution_mouse {49}
Executing ML code with '(mouseleft)'[324]. VIEW→ doc_for_ml_execution_cmds {46}

'(mouseleft)'[324] on DO→ <ml>
simply executes the ML expression, leaving the response on the stack.

'(mouseleft)'[324] on FN→<ml>
applies the ML expression as a funtion to the term at the point, whether in the same or a different window, and replaces the term by the term-valued answer. No-op if error. Also, the evaluation will be done in a context where the ML expression (current_object ()) evaluates to the name of the object containing the argument, not necessarily the same as the FN→<ml> button.

There are several variants of FN→<ml>:

FNcover→<ml> covers the argument with the value rather than replacing it.

FNpopup→<ml> opens a scratch window and puts the answer there.

FNquiet→<ml> puts the answer nowhere; the result can have any type, therefore.

DO→ <ml> simply executes the expression, leaving the answer on the stack. The object name at the point is available to it, however.

DOquit→ <ml> is like DO→ <ml>, but safe-quits the object containing the button.

To simplify the repeated application of these buttons, the command '(m-n)(m-n)'[323] will apply to the point the last function that was invoked by '(mouseleft)'[324] on FN→<ml> or its variants.

The variants of FN→<ml> and DO→ <ml> can be cycled through by repeatedly using the '(backspace)'[99] command, and FN→<ml> itself is inserted by 'mlbut'[152]. To toggle between the FN→<ml> family and the DO→ <ml> family, use '(c-a)(c-(backspace))'[101].

There is also an alternative display form for the FN→<ml> family of buttons, in which the user simply specifies a string to be displayed instead. Many of the buttons provided in this editing package are of this kind.
To convert FN→fu to display as FUBAR just use the command '(c-s)'[31], which will change the form to [string for button], then fill in this text slot with the desired display string. ('(c-d)'[81] will fill in the slot with the null string, which is the normal form for FN→<ml>). To convert back to the normal form, just go down to the text with '(c-j)'[62] and delete, say with '(c-a)(c-d)'[82] '(c-d)'[81].

Finding library objects. doc_for_finding_lib_objects {50}

1. Their are several commands for raising objects related to the point-term.

'vop'[46] will raise the definition of the operator of the point term, if there is one.

'vdf'[45] will raise the DISP object being used to display the point term, if there is one.

'vob'[44] will raise the object named by the point term, if any.

'show'[55] and '(cm-(mouseright))'[312] will usually raise a likely related object, most often doing a 'vop'[46] or 'vob'[44].

2. If you know a prefix or substring of the objects you want to find, you can try name completion on an object reference, say of form VIEW→ [ob name], which will return a bunch of similar object references.
   '(c-&)'[23] is name completion from a prefix; '(c-a)(c-&)'[24] is full name completion.
   VIEW→ doc_for_name_completion {41}

3. There is a utility ObTracker that tracks recently opened or closed objects. See the MainMenu {2}. To force a refresh of its status, '(mouseleft)'[324] on "Refresh" or anywhere outside the whole term, or use '(c-z)'[290] anywhere in the window.
   The main body of the term will be a listing of various objects, broken into classes.
   The user can also force the persistent tracking of specific objects by clicking on their names with Notice (and Forget undoes it).
   For explanation of annotations and options VIEW→ ob_tracking_MENU {13}.

4. The LibSearch button in MainMenu {2} and ObTracker will raise a macro for some ML code that will search the library for objects satisfying a condition expressed in ML; click on it now. Fill in the condition slot, and execute with '(c-z)'[290].
   You will also find a little menu for inserting terms commonly used in such searches.
   When the condition is applied to successive library objects, several ML ids will be bound to features of the object:

obtok	the object name as a token
kind	the object kind (possible values are in the menu)
term	the contents of the object
status	the verification status (possible values are in the menu)

   (Use the Buttons button on the little menu to get documentation on what they do.)
   The effect will be to raise a window containing references to the objects satisfying the condition.

5. The button WhereIsIt? in MiscEditMenu {61} (which can be raised with the MISC {61} button in MainMenu {2}) will show the names of objects that surround the object, or object referred to by the point, to which it is applied.

6. Theorems, operator definitions, or rules that mention one or more specific operators can be found with the 'thms'[52], '{defs≡abses}'[54], and 'rules'[53] commands. Definitions mentioning a given operator can also be found with '{(c-w)≡(m-w)≡(c-i)}'[12]'show'[55].

7. Lemmas with proofs explicitly citing a given lemma may be found with the WhoCites button in the proof editing menu PF, which is reachable from the MainMenu {2}.
   Proofs citing a given lemma can also be found with
   '{(c-w)≡(m-w)≡(c-i)}'[12]'show'[55].

This editor recognizes a special class of terms as references to objects, which expand in ML code to the token naming the object, and on which ‘(mouseleft)’[324] or ‘show’[55] will cause the referenced object to be viewed.

An obref always contains a pointer to the object, but there are three basic display methods (plus some combinations).
Here we discuss the variously displaying forms of object reference terms; we'll discuss their creation and editing in VIEW→ doc_for_ob_refs_editing {52}.

For this explanation, we have created a sample comment object VIEW→ sample_of_obref containing the following: Sample-Content.

1. The first way of displaying an object reference is to indicate the name of the object, and possibly a hint that it is an object reference.
   The default form ordinarily inserted via the ‘{obr≡obref}’[133] is exemplified above, namely: VIEW→ sample_of_obref. A trivial variation is →sample_of_obref.

2. The second form is to simply stipulate the text to be displayed such as THIS, possibly with some hint that it is an obref.

3. The third is to show the content of the object, again exemplified above, though not obviously so, namely: Sample-Content.
   There are some variations on this content display form.
   One is to show both content and name, thus: Sample-Content sample_of_obref

   When a content-displaying form is used on an object other than a comment object, the kind of object is indicated as well. When such a form is used to display a theorem, ‘(mouseleft)’[324] will expose the theorem content in a browsable form rather than raising the proof.
   Example: Thm* 0 = 0 sample_for_thm_obref
   The "*" indicates that the proof is good and complete.
   Clicking ‘(mouseleft)’[324] on this theorem obref will convert the form which merely shows the content, to one containing the content for perusal or cut-n-paste editing.
   Clicking ‘(mouseleft)’[324] again on the "Thm:" notation will restore the obref.

   (Further, to enhance readability, it is not necessarily the exact content that is displayed with these indirect content-displaying forms of obref, but rather, it is the result of applying the function ob_content_display_adjust to the object name.)

VIEW→ doc_for_ob_refs_editing {52} for how to edit obrefs.

Here we explain how to create and modify the various forms of object reference described in VIEW→ doc_for_ob_refs {51}.

Relevant commands: '{obr≡obref}'[133] '(csm-(mouseright))'[322] '(backspace)'[99] '(c-a)(c-(backspace))'[101] '(mousemiddle)'[313] '(mouseleft)'[324] 'show'[55]

BASIC

The basic obref creation command is '{obr≡obref}'[133], but many other commands will generate obrefs which can then be pasted into your documents.

Applying'{obr≡obref}'[133] to an empty term slot inserts "VIEW→ [ob name]". You may fill in this text slot however you like, but one special way is this is:

> Place the point cursor INSIDE the text slot where the obname is needed, then click '(mousemiddle)'[313] on whatever window you want this obref to point to.

An expeditious way to achieve this is simply to click '(csm-(mouseright))'[322] in whatever object you want to make a pointer to, and it will be inserted.
As usual in this editor, one moves between the various forms of obrefs using the toggling commands '(backspace)'[99] and '(c-a)(c-(backspace))'[101].
VIEW→ doc_for_toggling {42}.
The short toggle '(backspace)'[99] will cycle through the non-content-displaying methods, or through the content-displaying methods, whereas the long toggle '(c-a)(c-(backspace))'[101] toggles between the content-displaying and non-content-displaying forms.

Let's use the library object VIEW→ sample_of_obref in the examples below.

'(backspace)'[99] starting with VIEW→ sample_of_obref will cycle through these forms:

VIEW→ sample_of_obref	The basic form
→sample_of_obref	A slight abbreviation
[string]	The object name is hidden and the mnemonic string is displayed instead. If you insert the string empty, say by deleting the text then using '(c-d)'[81], it reverts to the →sample_of_obref form.

'(c-a)(c-(backspace))'[101] on any of the above forms will "switch tracks" to the following, among which you can toggle with '(backspace)'[99].
And from these, '(c-a)(c-(backspace))'[101] will switch you back.

Displaying the contents of the object in place:

Sample-Content

Displaying the contents of the object in place, along with the name of the object:

Sample-Content sample_of_obref

SPECIAL CASES

In an ML editing environment, VIEW→ doc_for_ML_editing {24}, references to theorems of the form "VIEW→ sample_for_thm_obref" will be automatically converted to the form "Thm* 0 = 0   sample_for_thm_obref" should the point be placed on them. The same goes for references to rule objects.

If the '{obr≡obref}'[133] command is applied to an instance of a non-system, i.e. a "math" operator, such as "∀x:ℤ. <prop>", then the content-displaying form of a reference to the definiton object will be inserting in it's place:
Def ∀x:A. B(x) == x:A→B(x)   all. The theory motivating this is that one often needs to create pointers to definitions when making documentation, and that the simmplest method is to piggyback on whatever methods one has for getting hold of operator instances in the first place.

---

doc_for_making_buttons {53}

Key commands: '{obr≡obref}'[133] '(csm-(mouseright))'[322] 'ibut'[139]
'(backspace)'[99] '(c-a)(c-(backspace))'[101]

Buttons are just terms you '(mouseleft)'[324] on to have some effect, usually related to where the point is. They are expected to work whether or not the point is in the same window as the button when you use it. So if you extend your editor to do this on a new class of terms, you've got yourself a new kind of button.

Buttons are displayed either show their function and content as explained below, or else are abbreviated to a mnemonic text such as THIS {53}. Here's a button to start a utility for changing generally how these buttons look: CHANGE. To make one of these abbreviated textual buttons from a complex-looking one, just use '(c-s)'[31] on the button then fill in the text slot that opens. To go back, just go into the text with '(c-j)'[62] and replace it with empty text, using '(c-d)'[81] twice.

Buttons can be embedded in any term. You can make your own menus or "palettes" (which connotes more flexibility) by simply collecting and creating various buttons into a term and arranging them however you like, typically with the word-processing constructs.
VIEW→ doc_for_WORDS_editing {29} VIEW→ doc_for_cut_n_paste {56}

In addition to the various special purpose buttons you will find among the menus, the most common buttons and the kind you are most likely to create, perform one of three kinds of action: view, insert, and call ML.
The three most common button forms are variants of:
VIEW→ [ob name], <term to insert>$^{+}$, and FN→<ml>

Each of these has variants in which a user-supplied mnemonic string is displayed instead of the button content; there are also operational variants.

Here are some references to documents about these basic button forms:

VIEW→ [ob name]   VIEW→ doc_for_ob_refs {51}
<term to insert>$^{+}$ VIEW→ doc_for_insert_buttons {80}
FN→<ml>   VIEW→ doc_for_ml_buttons {81}

button_conversion_menu {54}

Here are some buttons which you can apply to a button if you want to make a keyboard command that acts the same way.

BUT only simple buttons can be converted, namely, operator insertions, point-applications of ML ids, and object-viewing buttons. Error messages about these limitations appear in the LISP window.

Make GENERIC command.

Make ML environment command.

Make PLAIN environment command.

Make WORDS environment command.

SpecialChars {82}   UtilityForMoreComplexCommands {84}

Since keybard commands are overloaded, it is important that you select which environment you want the command to work in. VIEW→ doc_for_edit_context {22}

Limitations {55}

---

doc_for_button_conversion_menu {55}

Note about limitations of button conversion. (VIEW→ button_conversion_menu {54})

The editor is not yet set up to manage these commands like it does with the built in ones, so if you want to collect any commands for later use, say if you have to rebind all the commands for some reason, or want to use them in your other Nuprl sessions, you should save the commands you make in some library object, and sequence them with the sequencing constructor (<ml> ; <ml>) since executing the definition in ML is what binds the command to the name you give it.

There will be no automatic generation of the usual documentation for these commands. This may be rectified in the future.

Making buttons first is an easy way to make new commands.
VIEW→ doc_for_making_buttons {53}

doc_for_cut_n_paste {56}

The basic concepts for insertion and deletion, and many of the commands are explained in VIEW→ doc_for_term_insertion {19}.

Normally one simply copies terms operators from a visible sample using the mouse {'(mouseleft)'[324], '(m-(mouseright))'[321]} perhaps even using explicit copying commands {'(c-c)'[94], '(c-a)(c-c)'[95], '(c-f)'[96], '(c-a)(c-f)'[97],}, but sometimes one simply wants to get a copy of a term onto the top of the stack so that it may be inserted with '(c-p)'[87].
Here are some commands that temporarily replace the top of the stack:

```
    '(c-y)'[84]       copies the point.
    '(c-o)'[85]       copies the operator, deleting subterms and binding variables.
    '(c-a)(c-o)'[86]  is like '(c-o)'[85] but it also deletes parameter values.
```

Most other operations, including deletion with '(c-d)'[81], that change the point will leave the previous value on the stack temporarily.

The top of the stack is quite volatile, so for longer preservation, '(c-()'[89] will replace the top of the stack by TWO copies of the point, and '{(m-p)≡(cm-p)}'[88] will insert the second-from-the-top of the stack. You can pop the stack with '(c-))'[90], but usually it's not worth the trouble. The stack is normally kept rather small by periodically automatically chopping off the bottom. And instead of trying to manage several stack members during cut and paste, it is clearer to simply make scratch copies in new windows with the '{scr≡scratch}'[137] command which makes a scratch window containing the point value. Then just cut and paste from the scratch windows.

BUGS: The stack is too quickly modifed when the point is on an inference step, so cut and paste must be accomplished without explicit stack manipulation commands. Commands '(c-c)'[94], '(c-a)(c-c)'[95] and '(m-(mouseright))'[321] are recommended for copying an inference step on top of another.
VIEW→ doc_for_proof_refinement2 {27}

Two especially handy commands are '(c-u)(c-(space))'[78], which promotes the point by deleting the parent and siblings; and '(c-u)(c-d)'[80] which deletes the point, and if it has only one sibling, promotes that sibling by deleting the parent.

For the rest, review the individual commands with the following:

DO→ Review cmds for group: "Cut&Paste"

doc_for_deptypeinsert {57}

There is a standard method for flexible entry and editing of dependent type operators in Constable's standard notation.
We'll use dependent function in this explanation.

Constable's notation for dependent function space is
 x:A→B(x),
with the special case of type independence notated
 A→B.

The command for inserting this operator is '>'[116].
Normally, insertion inserts <Type>→<Type> in left-mode, and [var]:<Type>→<Type> in right-mode, and replace-mode insertion inserts <Type>→<Type>.

But if the point is at an instance of ([var]:<type> ?), which is inserted by ':'[77], left-mode insertion will build the dependent form using the typing indicated.
Furthermore, insertion with ':'[77] is specialized in anticipation of this.
Examples: x:A→<Type> can be created by entering
 'x' ':'[77] 'A' '>'[116] in a plain slot, or
 'A' '>'[116] or '>'[116] 'A' into the slot of x:<type> ?,

To get at the binding var position which is hidden in
A→B, '(c-s)'[31] will temporarily reveal the binding position, and will delete it if it is null.

It is easy to null-out the binding var position in a slot simply by using
'(c-u){(tab)≡(return)}'[102], which will then skip to the next slot if any.

\---

Posting, Quitting, and Deleting Objects. doc_for_posting_objects {58}

'(c-q)'[286] '(c-a)(c-q)'[289] '(c-v)'[284] '(c-a)(c-v)'[285] '(c-z)'[290]
'deleteme'[287]

Changes you make in an object window are not posted to the object itself until you request it.

DO→ Review cmds for group: "Posting"

These raise worksheets: io_menu {59}

ListKnownTheories

ListObsOfFile

LoadFileOfObjects

ScanFileOfObjects

LoadTermFromFile

LoadTextFromFile

LoadMLfromTextFile

EvalTermOfFile

CompareFileOfObjects

FindLoadDir

ListAllLoadDirs

DumpTerm

DumpMLexpression

DumpObjects

PrintObSet

FileNameTips

Other I/O related menus:
SNAPs {85}
Theories {86}
Saving&Recovery
Tailor this Editor {7}
Printing Editor Doc {6}

FindSame NextMark ClearMarks QUIT edit_find_menu {60}

FN→edit_search_term <ml>:term→bool

FN→{Replace lastpred terms ↑} **λ**r.higher_map (**λ**t.if lastpred t then r else fail)
Replacement Term: ⌈<term (hungry)>⌉

lastpred$^+$ Eq

EditMatch pat|pat *RECURSE*$^+$ *ANY*$^+$ [string]:*$^+$ explode

VIEW→ doc_for_edit_match {18} VIEW→ doc_for_term_search {43}

AdjTypeDisp	AdjustOp	ApToggle	miscedit_big_menu {61}
Burst	CMDedit {84}	Cascade	
Comp120	DiffPost	Diff	
Display	Enum	Eval	
Expand	Fill	FreeVars	
Hide	LevelInfo	LoadDir	
MLeval	MakeReplOp	MakeWFthm	
Mark	Norm	ObrefsOf	
OpsOfTerm	PadSeq	ParcelSeq	
ReHide	ReLayout	ReverseSeq	
Scratch	Sort	Step	
TermSize	ThmsForOp	UNenum	
UNpadSeq	Unhide	WorkDir	

WhereIsIt? MoveIfMust SPECIAL

DEBUG mledit_big_menu {62}

MLbootDir Precedence MLfiles

MLediting {24} ML manual (postscript)
TacticNames   Tactic manual (postscript)

General {88} Term Primitives Lib {89}
MLtypes {90} ListGenericMLops

MLmacro$^+$ ml$^+$ letrefASSIGN AddRefCites

ShowMLobIdsFor MLfreeIds

WhoCalls (Fully)   CallsWhom

ReExecutePostedDefOfMLid

CompareDefs UpdateDef DeleteDefOf

InsertCodeIntoMLobject

AddToMLcache RemoveName RefreshCache {91}

RecentlyDefinedMLids

linear   consistent

breakBeforeBreak breakAfterBreak

Indirect   CreatePassiveIndirect
LabeledContent IndirectModification

[text to emphasize]$^{+}$ PostscriptRef

<words table>$^{+}$ Tabulate
<words to pad>$^{+}$ PadSeq   UNpadSeq

Norm Burst Sort Enum UNenum Parcel20
ConvertTextToWords PushDownJams Buttonize

UpCase DownCase SpellCheck

ShowFont(nuprl-13)

$\mathbf{\Sigma\Pi\Delta}\times\forall\exists\cap\cup$ $\mathbb{UPBNZQRC}$ $_{\mathbf{0123}}{}^{+-0}{}_{\boldsymbol{abcqz}}$
$\downarrow\uparrow\leftarrow\rightarrow\rightarrow\leftarrow\Longleftrightarrow\rightleftharpoons$ $\wedge\vee\neg\neq\equiv\geq\leq\supseteq\subseteq\supset\subset$ $\boldsymbol{\alpha\beta\gamma\delta\in\lambda\pi\rho}$
$\cdot\div\pm\oplus\otimes\boldsymbol{\infty\partial}\int\vdash{}^{\ulcorner\urcorner}$

COMPRESS REVERSE

RELIST1 RELIST2 RELIST3 RELIST4

AllowInlineBreaks

How to find documentation and definitions for global ML ids.   doc_for_ml_doc {64}

Name completion with ‘(c-&)’[23] and ‘(c-a)(c-&)’[24] is helpful for finding the right name of an ML global id. VIEW→ doc_for_name_completion {41}

To find the type of an ML id, which should be an instance of [text], just execute it in place with ‘{(m-z)≡(cm-z)}’[291]. VIEW→ doc_for_ml_execution_cmds {46}

ML code comes in three forms:

structured code in ML objects (VIEW→ doc_for_ML_editing {24}),
unstructured code (text) in ML objects, and
text files from disk.

There is an object →ml_files which contains the names of the standard ML files for Nuprl; ‘(mouseleft)’[324] on one of these "file pointers" will start up a fresh viewer on the file via the shell command "view".
(For future reference you can find a pointer to →ml_files in an ML edit utility object which can be raised with the "ML {62}" button in MainMenu {2}.)

To view the definition of an ML global, use the ‘show’[55] command. If it’s defined in an ML object of the library using structured ML, then a copy of the definition will be raised in a scratch object, along with a pointer to the ML object; if you want to view the ML object immediately, use ‘findml’[153] instead of ‘show’[55]. If the ML id is defined in an ML object of unstructured text, it may or may not be found, but if it is, the ML object will be raised. If it can’t find the definition, it will raise the object →refresh_mlob_def_cache_menu {92}. That object contains a button for grepping the standard ML files for the last ML id on which ‘show’[55] was attempted, as well as a pointer to →ml_files described above.

To find other ML globals whose definitions cite a given ML global, use the WhoCalls button in the ML EditMenu {62}, which is reachable from the MainMenu {2}.

There is further documentation on tactics in Paul Jackson’s manual for Nuprl. The button TacMan will ghostview the part of the manual pertaining to standard tactics and the so-called theory structure.
(For future reference you can find this button in an ML edit utility object which can be raised with the "ML {62}" button in MainMenu {2}.)

There is also a manual for the ML programming language: SHELL ghostview ml-manual.ps {nuprl/doc/ml}

Keyboard commands, which we shall consider to include mouse commands, are referred to in documentation in such a way as to suggest the actual characters with modifiers that you enter to invoke them. Examples are

‘all’[115] ‘>’[116] ‘=’[112] ‘{(DOWN)≡(c-u)(c-j)}’[60]
‘(mouseleft)’[324] ‘(c-(mouseleft))’[310] ‘(m-(mouseleft))’[309]
‘(m-(mousemiddle))’[315]
‘(cm-(mouseright))’[312]

The shift, control, and meta modifiers are indicated by c,s, and m, and synonymy of character sequences is indicated by {<words>≡<words>}.
Other special characters may be found in the edit menu SpecialChars {82}.

Invoking Commands.

To invoke a command, enter the sequence of characters indicated in the nomenclature for the command. An identifier or numeric string that is not bound to a keyboard command is interpreted as inserting the identifier or numeric string in a way determined by the editing context (see EditContext {22}). To avoid amiguity, commands that end with an identifer or digit must either be terminated with ‘ ’[2] or immediately followed by a command starting with a non-identifer character. Here are the characters considered identifier characters:
‘abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_’’[3]

Commands starting with ordinary printing characters cannot be issued when the Nuprl cursor is located inside text, since ordinary characters in text are taken as self-inserting. Nor can (rubout) used inside text except to delete the character in front of the cursor.

‘(c-g)’[1] clears the command scanning buffer so that command entry can be aborted. Unfortunately, clicking the mouse in another window also clears the buffer of everything preceding the mouse command.

Changing keyboard commands.

Keyboard commands have an internal name and an external binding which is displayed in the notation referring to the command. VIEW→ doc_for_edit_cmd_review2 {70} for how to change keyboard command bindings.

The Lisp window is used constantly to emit short advisories about your Nuprl state, and should be placed where it is easily observed.
The messages may be more readable if you create the LISP window in a way that makes the nuprl-13 font work in it.
You might use "xterm -fn nuprl-13" to do this.
You may also need to enter commands into it to force a halt or recover from an error.
For advice on using the Lisp window, please see
~nuprl/users/sfa/nuprl_trouble_lisp or ~nuprl/users/sfa/nuprl_trouble_allegro.

Use of this editor commonly causes numerous windows to be raised that will need to be moved or resized.
Resize to reveal terms elided to "...".

It is highly recommended that you adopt some mouse commands for controlling X-windows by pointing into the window itself rather than having to point at the title bar of the window.

It is also useful to collect your Nuprl window icons under a single icon manager sorted alphabetically.

If you are using TWM as your window manager, you can easily add these capabilities by appending to your .twmrc file. Here is a button that will do this for you. You should examine the appended file to see how the new features work.

```
DO→ SHELL mv .twmrc HOLD.twmrc {~} ;
    SHELL chmod -w HOLD.twmrc {~} ;
    SHELL cat HOLD.twmrc ~nuprl/nuprl4/doc/admin/append.twmrc >.twmrc {~}
```

In order for this to take effect, restart your TWM window manager or, if you have a TWM menu command for f.twmrc, you could execute
it in order to give your new .twmrc effect.

This editor is designed to edit entire proofs as terms. For advice on editing proof terms, see ProofEditing {25}. Here is a sample proof term, which includes an unproved leaf.

```
 1. A : Prop
 2. B : Prop
 3. C : Prop
 ⊢  (A ⇒ B ⇒ C) ⟺ (A & B ⇒ C)  by  Analyze

  \
   ⊢  (A ⇒ B ⇒ C) ⇒ A & B ⇒ C  by  UnivCD ...w

   4. A ⇒ B ⇒ C
   5. A & B
   ⊢  C  by  Analyze5

   6. A
   7. B
   ⊢   by  BackThru: 4

    \
     ⊢  A  by  Auto
     ---
     ⊢  B  by  Auto
   ---
   ⊢  (A ⇒ B ⇒ C) ⇐ (A & B ⇒ C)  by  UnivCD ...w

   4. A & B ⇒ C
   5. A
   6. B
   ⊢  C  by  <TACTIC>
```

For each assertion of the proof, the proofs of its subgoals is listed beneath it. If there is only one, it is just listed below in line. If there are more than one then they are shown like this:

```
 Goal  of proof  by  Tactic code for generating subgoals

  \
   Proof of subgoal 1
   ---
   Proof of subgoal 2
   ---
   Proof of subgoal 3
```

CONTINUE {68}

(continued from VIEW→ doc_for_pfterms {67})

Navigating Proof terms.

It is suggested that you try these commands on the sample proof in VIEW→ doc_for_pfterms {67}.

The units of assertion are called "sequents" and consist of a "conclusion" and possibly some "hypotheses" or "declarations".

```
    1. A : Prop
    2. B : Prop
    3. C : Prop
    4. A ⇒ B ⇒ C
    5. A & B
    ⊢  C
```

This sequent has conclusion C, three declarations (of variables A, B, and C), and two hypotheses, A ⇒ B ⇒ C and A & B.

To save space and reduce complexity for the user, hypotheses inherited by a subgoal from its parent are normally elided. In some methods of display, the conclusion is elided too if it's inherited.

Sometimes a section of the proof will be elided and displayed as ***; '(mouseleft)'[324] will reveal the elided term.

To zoom in on a subproof, use '(mouseright)'[317] on some part of the proof specific text of the goal you want to focus on (as opposed to the conclusion or hypotheses themselves). To zoom back out you may '(mouseright)'[317] on the entire visible proof term, or the highest goal.

When you zoom in on a subproof, the address of that subproof is revealed, and looks like "top 1 2 1 1" indicating the path down from the top goal. If the point is located on or within a proof term, then '(mouseleft)'[324] on "top" or one of these digits will navigate to the node at that proof address. (Thanks to Pavel Naumov for this idea.) This works even if the point is in one window, and the address notation is in another. This helps coordinate navigation when one has two copies of a proof.

When 'show'[55] or '(cm-(mouseright))'[312] is used on proof term step, the tactic will be analyzed into simpler tactics or even primitive rules, and run in a new window.

doc_for_edit_cmd_review {69}

Normally, a keyboard command is associated with some documentation and an object specifying what keyboard string is bound to that command.
Here’s an example of a "review" for a keyboard command:

‘all’ kbd_tt_all <CHANGE> (plain, insert) ∀[var]:<type>. <prop>

indicates that the command named kbd_tt_all is associated with the keyboard string "all", and is a standard insertion command that is invoked only in a PLAIN environment, and inserts the indicated term. Generally, if the command behavior is conditioned on one of the standard edit contexts VIEW→ doc_for_edit_context {22}, that will be indicated. If the kind of behavior is one of a standard kind, that will be indicated:

insert   means it is a standard insertion command. A sample of insertion is shown. VIEW→ doc_for_term_insertion {19}

apply   means a certain ML function will be applied to the point, replacing it by the resulting term. If there is a comment documenting that function, it will be shown in the command review.

hideapply is like apply, but the point will be left intact, and the answer will be returned on the stack instead.

There may be multiple behaviors or qualifications associated with a command, and several such entries may then be listed.

NOTE: the standard bindings don’t combine control- or meta- with shifted PRINTING characters. This allows us to indicate the unshifted printing character by showing its shifted companion instead, which is often better mnemonically. So, (c-?) may would actually indicate (c-/).

Normally the indicated keyboard string is the one the command is bound to, but it is possible to change this string before the actual binding is effected.
You could go find this binding object and edit it directly, but this review utility is designed to simplify that task.

‘(mouseleft)’[324] on a review will record the command in the list of "noticed" commands, which can then be reviewed as a class using the button: Review

If you want to change the keyboard bindings for commands,
VIEW→ doc_for_edit_cmd_review2 {70}.

(continued from VIEW→ doc_for_edit_cmd_review {69})   doc_for_edit_cmd_review2 {70}

How to change keyboard bindings for commands for a review object, which is an instance of:

'(No object specified)' [kbd cmd name] <CHANGE> <DOC>

Normally the indicated keyboard string is the one the command is bound to, but it is possible to change this string before the actual binding is effected.
You can find this binding object with the 'vob'[44] command on the review term, and edit it directly, but this review utility is designed to simplify the task.
If you '(mouseleft)'[324] on the CHANGE slot above, it will be filled with some options:
→Okay |→Remove |→Restore |[word]

If you execute '(c-z)'[290] on the command review expression, the binding object will be updated according to how this slot is filled.

- →Okay will leave the binding object as is. Same as if you leave the original slot empty.
- →Remove will clear the binding object, which will mean the command will have nothing bound to it when you effect the actual bindings.
- →Restore will restore the binding object to the value it had the last time actual bindings were effected.

Otherwise, the slot should be filled in with a keyboard string description, and this will be stored to the binding object.
You can either fill in the [word]-slot with the description directly, or you can use the special character edit utility: VIEW→ edit_cmd_special_char_menu {82}.

If you need to start over, just delete with '(c-d)'[81] whatever is in the CHANGE slot, and the options will be inserted again.

Remember, the new binding object contents will show in documentation even before you make the actual bindings occur, so don't forget what your current keyboard bindings are.

You should make sure you are happy with the way the keyboard strings are overloaded.
Here's a button to review overloads: ReviewOverloadedKBDcmds

When you are ready to effect the bindings, this button will do it: ReBindAllKBDcmds

ForceQUIT

thm_viewing_selection_menu {71}

Choose your default THM viewing method. You may use either the standard refinement editor RED, or you may edit terms representing entire proofs.

Use RED? Yes

Edit via Proof Terms? Yes

You may save you preference for future sessions with this button:
SAVE THM viewing preference
---

doc_for_ML_scanner {72}

Unstructured ML and the underlying scanner. (also VIEW→ doc_for_ML_editing {24})

The underlying ML interpreter reads a string of chars and terms. When ML is invoked on terms in the system, it converts the term to a character/term sequence for the underlying interpreter. It fully expands operators in the term until there are none left attributed with one of the conditions "Macro" or "ml". So "structured" ML code is built from operators used as macros to expand to the appropriate text.
VIEW→ doc_for_ops_as_macros {93}
Then instances of [text] terms are passed as the characters filling them, and the binary operator: <pseudo text><pseudo text> is used for concatenation in the sequence. Terms with other operators are passed as themselves. These operators for representing text strings and concatenation can be passed as literals by enclosing the terms in ⌈<term (hungry)>⌉.

NOTE: You can also forestall the creation of structured ML and enter a text string to be parsed by ML by using the ‘(c-t)’[109] command. In an ML environment, ‘(c-t)’[109] will insert [ml text] into which you can enter text. Whenever the point is placed on this form of term, typically by moving up to it from the text with ‘{(c-k)≡(cm-k)}’[64], the editor will attempt to parse it and convert it to structured ML code. If that fails, an error message will be inserted, and the user must alter the text (‘(c-d)’[81] to delete the text).

Editing Let Forms. doc_for_ML_editing_let {73}

'let'[244] will insert

  let[,rec,ref,type] <ml> ≡ <ml> in <ml>

or

  let[,rec,ref,type] <ml>≡ <ml> ;;

depending on whether you are in a context that requires an ML command as opposed to an expression, mainly whether the point is on one of the immediate subterms of

  <ml> ;; or <ml cmds>
            <ml cmds>.

If you insert on top of a term, it will go into the place specifying the binding value in normal left-to-right mode. If you insert the local-let form in right-to-left mode, the term will go into the body of the let-form instead. The left/right insertion mode is toggled with '{(c-w)≡(m-w)≡(c-i)}'[12].

'glet'[242] will insert the global let-form directly.

You may fill in the text slot indicating the kind of declaration you are making; '{(tab)≡(return)}'[73] will fill the text slot with the empty string, indicating the ordinary binding, so
'let'[244]'{(tab)≡(return)}'[73] is the most common combination.

'(backspace)'[99] when the point is on the Let form will cycle through the several choices permitted for this string.

You can also toggle between the global and local Let forms with '(c-a)(c-(backspace))'[101]. This is particularly useful when you want to develop or modify a function before binding it to a global variable.

Linear forms for Global-Let and Zooming.

In order to save visual space, global-lets have two forms. One is displayed linearly, and will elide the body if necessary. This linear form is considered the normal one for completed global definitions, since we tend to have a lot of them in one object. You can force this form of declaration to be fully shown throughout window with the '(c-s)'[31] command, but that can be a problem when editing an object with more than a couple of these.

The normal way to convert from the linear to the full-display form of global-let is simply to zoom on the let-form, whereas unzooming will convert the form back to linear. So if you wind up with a non-linear form of global-let in a larger term and want the linear form, just zoom on it then unzoom.

The usual zooming commands will work, but also '(mouseleft)'[324] on a linear-form of global let will do the zoom. VIEW→ doc_for_point_focus {11}

In ML, the If construct has several forms permitting the expression of loops.

The basic form is

```
if <ml> then <ml> else <ml>
```

which can be inserted in ML mode with the ‘if’[175] command; insertion of the current point term is either into the Then part or the Else part, depending on the setting of the left/right-mode.

The variant forms are not essential, but are edited as follows.

If the If form is nested through the Else place, then the "else" particle can be toggled with the ‘(backspace)’[99] command. Toggling like this if the Else place is empty will insert another If form before removing the "else" particle.

The presence of the "else" particle becomes significant only in order to determine the scope of a loop construct. The basic form can be toggled with loop forms thus:

```
if <ml> loop <ml> else <ml>
if <ml> then <ml> loop <ml>
if <ml> loop <ml> loop <ml>
```

are variants indicating when looping will occur from the branches.
In normal left-to-right mode, ‘(c-a)(c-(backspace))’[101] toggles the then/loop particle, while in right-to-left mode, it toggles the else/loop particle.

THM objects versus proof terms. doc_for_thm_posting {75}

The mechanism the system uses for certifying proof legitimacy is mediated by THM objects. THM objects are marked as complete and correct, partial and correct, or bad.

Bad THMs are those in which some tactic fails or the goal has free variables.

Partial THMs have correct steps, but have subgoals that have not been proved.

References to THM objects having the form "Thm* $\forall$i:$\mathbb{N}$. i$\geq$0 nat_properties" will include the "*" character to indicate that the THM object is good and complete.

To Store a Proof to a THM object, one first connects the THM object with the proof term with the following operator

Proof of fubar Ob Not Found: $\rightarrow$fubar

<pf term>

where the name of the THM object, say fubar, is in the string slot and the proof term goes in the term slot. The content of the THM object, if there is any, will be shown but not editable, and will display as ?fubar?.

To store the proof to the THM object, locate the point on the Proof-of operator and execute '(c-a)(c-v)'[285] (which is the generic command for posting data to objects indirectly, which is to say, to objects other than the one being edited).

When a THM object is viewed or created, this Proof-of operator is automatically created for you. The PF menu accessible from MainMenu {2} also contains a button for inserting it.

---

doc_for_abs_verify {76}

When an operator definition or declaration, an ABS object, is "verified" with the '(c-v)'[284] command, the editor checks the existence and contents of various related objects for coherency with the ABS object, and may raise a utility for modifying or adding related objects such as display form specs (DISPs), well-formedness theorems (THMs), and "pseudo-step" specifications which are mainly to expand recursively defined operators.

For a newly defined operator, the utility has buttons for defining a new display form and, normally, a new well-formedness lemma.
If your definition is recursive, you should allow the pseudo-step spec to be created.

If you change a definition, you will get a number of buttons for updating the auxilliary objects.

```
<specs>                                                   dfedit_big_menu {77}
<specs>+  Spec  Ignore   DformSpecsForTerm  HELP {34}

LeftAssoc  RightAssoc ListAllPrecObs

StdIter StdIterExc StdIterNub StdIterNubExc
StdQuant StdMLinfix

<Formats><Formats>+   <child attrs>, <child attrs>+

ChildMenu DFchild   * |Other+  Iparms+

Enum    UNenum    ListAllPrecObs

<attrs>; <attrs>+ NOattrs+          <term to insert>+ CondsMENU
*Open form*+      InsertCmd:[name]+ HiddenSlotOK+     ShowGrandchild+
InvisibleOp+      InheritParens+    PassThru+         NotAtPoint+
=[obj].[name]+    =apply.standard+  =[name]+          =FIRST+
Name:[name]+      Family:[name] +   member:<Fams>+    <Fams><Fams>+

{[hard]}+      [[break]]+     {Space}+
<disp>←CONSIS+ <disp>←LINEAR+ <disp>←MARGIN+

<Format>+

Move DISP to User Redisplay Area. SAVE redisplay objects.

ConvertToStandardForm
---
```

doc_for_edit_choice {78}

Sometimes the editor will provide a list of options in place, by presenting those options in a sequence connected with <term> |<term>.

If you click on one of the options with '(mouseleft)'[324], or locate the point to the option and hit '(c-u){(tab)≡(return)}'[102], it will replace the whole sequence.

You can also narrow the choices by simply entering some characters as an identifier or numeric sequence, and this will shrink the sequence to those containing that substring in their display.

If you hit '(c-u){(tab)≡(return)}'[102] on an option sequence, it narrows to the shortest strings, or selects the first option if they're the same length.

You can move down into a (right nested) sequence of choices with '(c-j)'[62] and '(cm-m)'[93], and move between choices with '(c-l)'[66] and '(c-h)'[65].

'(backspace)'[99] on a sequence of choices will rotate the sequence, and reversing the left/right mode will rotate the other way (not sticky).

In ML expressions, an edit choice with multiple options is evaluated as if the first option were selected (just look at the definition of <term> |<term>).

ForceQUIT forgetall_menu {79}

Forget ALL

The command ‘ibut’[139] inserts <term to insert>$^+$. doc_for_insert_buttons {80}

The command ‘(backspace)’[99] toggles between <term to insert>$^+$ and <term to insert>$^-$.

‘(mouseleft)’[324] on <term to insert>$^+$ inserts the enclosed term at the slot, as specified in VIEW→ doc_for_term_insertion {19}.

‘(mouseleft)’[324] on <term to insert>$^-$ is similar except that it reverses the left/right direction first. (See the ‘{(c-w)≡(m-w)≡(c-i)}’[12] command.)

If you want to display the button as a mnemonic string, you should locate the point on the whole <term to insert>$^+$ button and use the ‘(c-s)’[31] command, which will convert the display, and then you should enter the display string you want.
If you use the empty string, say by hitting ‘(c-u){(tab)≡(return)}’[102] on the text-slot prompt, the form will revert. You can also clear text with the ‘(c-a)(c-d)’[82] command when the point is in the text.

To insert FN→<ml> use the command ‘mlbut’[152]. doc_for_ml_buttons {81}

‘(mouseleft)’[324] on FN→<ml> will apply the enclosed function to the point, and replace it by the result.

The following variants can be gotten by toggling with ‘(backspace)’[99]:

FN→<ml> FNcover→<ml> FNpopup→<ml> FNquiet→<ml>

VIEW→ doc_for_ml_execution_cmds {46} for detail of how they execute.

You can get DO→ <ml> from any of these with the ‘(c-a)(c-(backspace))’[101] command.

‘(mouseleft)’[324] on DO→ <ml> will execute the enclosed ML expression. The variant DOquit→ <ml> toggles via ‘(backspace)’[99] with DO→ <ml> and attempts a safe quit of the object containing the button.

To display one of these buttons as a mnemonic string, use the ‘(c-s)’[31] command on it, and fill in the text slot with the desired string. Emptying this string will revert the form. ‘(c-u){(tab)≡(return)}’[102] in the empty text slot, or ‘(c-a)(c-d)’[82] in the text slot empty or not, will make the string empty.

Various ways of executing ML commands are documented in VIEW→ doc_for_ml_execution_cmds {46}.

[word]$^{+}$ <words><words>$^{+}$ edit_cmd_special_char_menu {82}

IndirectWord {<words>≡<words>}$^{+}$

You can toggle c,s,m with the cmds: ‘(c-(backspace))’[303] ‘(s-(backspace))’[304] ‘(m-(backspace))’[305]

or you can use these buttons to toggle:
c- s- m-

(mouseleft)$^{+}$ (mousemiddle)$^{+}$ (mouseright)$^{+}$

(space)$^{+}$ (return)$^{+}$ (tab)$^{+}$ (newline)$^{+}$ (linefeed)$^{+}$ (page)$^{+}$
(backspace)$^{+}$

(DOWN)$^{+}$ (UP)$^{+}$ (LEFT)$^{+}$ (RIGHT)$^{+}$

(c-a)$^{+}$ (c-u)$^{+}$ g^{+} {(c-n)≡(m-n)}$^{+}$ (m-q)$^{+}$ {Extender prefixes }

HELP & Advice {83}

---

doc_for_cmd_string_edit {83}

The representation for command strings used by this editor is as a word or sequence of words jammed together with <words><words> with certain characters represented by special strings rather than themselves.

Also, the following characters must be preceded by \ in order to represent the character: ( ) ’ = \ %

You should avoid using (m-x) and (c-x) in command strings.
You should also try to make commands that are used heavily in conjunction with the mouse be easily entered with the other hand.

Click HERE {82} for a utility for editing these command string representations. In it you will see various buttons for inserting string representations. Most are obvious; they represent individual characters, including combinations of control, shift, and meta keys.

Near the bottom are the extender prefixes, which are not essential, but can be handy to use since you may well want to treat them systematically, as the default settings for kbd bindings do.

You can specify the equivalence of two cmds, or parts thereof, by joining them with {<words>≡<words>}.

The least obvious device is inserted by the button IndirectWord; it is a special obref to an object which will use the contents of that object as part of the word string.

```
ButtonConversionMenu {54}                                    new_cmdedit_menu {84}

DEBUG  StdEdCmdDefs  RefreshCmdDocCache  CmdReview

EdCmdDoc <name>+
Edit Cmd Def Stamp <term (in ml)> for <ml>-
EdCmdDocGroups <words> for <ml>-
Left-Hand <kdb cmd> Version Recommended+

EdCmd <name> :<conds> == <body>+
EdCmd <name> :{@}<conds> == <body>+
EdCmd <name> :"PLAINenv"<moreconds> == <body>+
EdCmd <name> :"MLenv"<moreconds> == <body>+
EdCmd <name> :"WORDSenv"<moreconds> == <body>+
IfEdCmd <name> :<conds> == <body>+
WhileEdCmd <name> :<conds> == <body>+
RepeatWhileEdCmd <name> :<conds> == <body>+
{Rarer Forms ↑}
@Message([text])+ ReplaceSpacesAndCheck

@StackApply( edit_display_generic_help_tip
            |edit_display_generic_button_help_tip
            |edit_display_for_view_via_obref
            |edit_display_safe_quit_failure_help
            |edit_display_pointsuppression_help
            |edit_display_deepclean_help_tip
            |edit_display_cover_deletion_tip
            |edit_display_inert_ob_ref_help)+

EdCmd <name> :<conds> ==  <body> With Scout: <scout body>+

    [scout-cmd-prefix]<words>+

words_charinsert_env_edit_cmd_def{[string]:s}(<term>; <term>)+

IfMLcondEdCmd <name>:[ml fn name] == <body>+

TryUpToEdCmd <name> :<conds>+

CycleEdCmd <name> == <body>+

NameCompletionEdCmds for <condition> with
    SimpleCompletion Function: <ml text>
      FullCompletion Function: <ml text>
       Cache Refresh Function: <ml text>
    +

Replace&FinishEdCmd <name> :<conds> == ob:[ob name]+

(In)depTypeInsertEdCmd <name>
  with Dependent form   ob:[ob name]
  and  Independent form ob:[ob name]

DeclKbdCmd <name>+
```

```
make_edit_env_edit_cmd_def_for{[token]:t}(<term>; <term>; <term>)+
MakeEnvEdCmd <name> for <env abs>+

FrameEdCmd <name> with disp: [framedf]+

[[internal macro name]]+
*[primitive cmd name]*+  PrimCmds
{<words>≡<words>}+ IndirectWord SpecialChars {82}
KBDcmd          CreateKBDcmdOb
ListAllKBDcmds  ShowKBDbindingsFor
FindCmdDefsFor  FindCmdThreadsFor

{popush}+ {[abstract cmd name]}+

@Insert{ob:[ob name]}+    @TextInsert(<word>)+ @ObInsert(ob:[ob name])+
@TextObInsert(<word>)+ @Repl&Fini{ob:[ob name]}+ @SimpleApply(<ml text>)+
@Apply(<ml text>)+        @HideApply(<ml text>)+ @StackApply(<ml text>)+
@ViewOb([word])+ ob:[ob name]+             pushtext:[word]+

DefineTheseCmds    Define EdCmds with <MacroDefs>+
---

ClearSnapFile DO→ set_snapshot_file "∼/snapshot.text"              snapshot_menu {85}
SNAPSHOT Term
SNAPSHOT Horizontal Line
SNAPSHOT PageBreak
SNAPSHOT Window        MakeNodeCopy PrintProof
SNAPSHOT Listing

DO→  chosen_latexize_file |latexize_file_woambles "∼/snapshot.text" "∼/snap.tex"
SHELL latex snap {∼}  SHELL xdvi snap & {∼}
SHELL dvips -f snap.dvi >snap.ps & {∼}
SHELL ghostview snap.ps & {∼}
SHELL lpq {∼}  SHELL lpr snap.ps & {∼}
SHELL xterm -e $VISUAL snap.tex & {∼}     SHELL xterm -e $VISUAL snapshot.text & {∼}
SHELL spell snapshot.text {∼}

VIEW Listing    BurstObject EmptyBURSTs
```

QUIT theory_system_menu {86}

Depict Standard Theories

ListTheories with Load Status

Registered Theories (as buttons).

```
DO→ convert_theory_to_segment "[mlstring]"

DO→ dump_theory "[mlstring]"

DO→ add_theory_delimiters "[mlstring]"

DO→ let theory_name ≡ "[mlstring]" ,
        filename ≡ "[mlstring]" in
     set_theory_filename theory_name (complete_file_name filename)
```

Loading other theory files←

<ml> [label]:?DEBUG <term>+ ml_debug_menu {87}
<ml> [label]:REPORT <fun>+
(Remove Printers)

in_n_out←

VIEW→ mltracing

Here are some buttons for functions from VIEWprlml→ general   general_ml_menu {88}

max+ min+ sum+ max_value+ list_max+ abs+ sign+ gcd+ lcm+

split+ firstn+ nthtl+ split_lastn+ lastn+ last+
remove_lastn+ remove_last+ replace_nth_by+

search_list+ rev_search_list+ position+ find+

divide_list+ filter+ remove_if+ is_prefix+ is_suffix+ is_sublist+
remove_prior_duplicates+ remove_later_duplicates+
longest_prefix+ longest_suffix+

mk_set+ insert+ intersection+ diff+ union+ remove+
subset+ all_distinct+

replicate+ upto+ number+

quicksort+ merge+ mergesort+

add_to_alist_start+ add_to_alist_end+ apply_alist+
rev_apply_alist+ is_bound+ update_alist+
multi_update_alist+ modify_alist_entry+ modify_or_add_alist_entry+
remove_alist_entry+ totally_remove_alist_entry+ move_alist_entry_to_end+
merge_alists+ priority_merge_alists+ group_alist_entries+

char_lt+ tok_lt+ string_lt++ list_lex_lt+

upcase_string+ downcase_string+ is_substring+
---

lib_ml_menu {89}
Here are some buttons for functions from VIEWprlml→ lib, plus some defined in lisp.

new_load2+

destruct_rule_term_tree+ rule_term_tree_of_thm_object+
update_rule_term_tree_of_thm_object+

is_lib_member+ is_object+ kind_of_object+

object_body_to_term+ term_of_object+ term_to_object_body+ set_term_of_object+
library+ lib_segment+

goal_of_thm_object+ type_of_extract+

abs[,rec]type <ml decls> with <ml decls> in <ml>$^{+}$ mledit_types_menu {90}
abs[,rec]type <ml decls> with <ml decls> ;;$^{+}$

<ml> ≡ <ml>$^{+}$ <ml decls> and <ml decls>$^{+}$

<ml> [type op]$^{+}$
<ml>,<ml> [type op]$^{+}$
<ml>,<ml>,<ml> [type op]$^{+}$

<ml>→<ml>$^{+}$ <ml>#<ml>$^{+}$ <ml>+<ml>$^{+}$ <ml> list^{+}

ForceQUIT edit_refresh_ml_name_cache_menu {91}

Refresh ML Name Cache? YES.

It enables
1. ml name completion via '(c-&)'[23] and '(c-a)(c-&)'[24]
2. fancy lambda- insertion for ml via '{la≡\\}'[110]
3. fancy in-place evaluation via '{(m-z)≡(cm-z)}'[291]
4. locating definitions of identifiers in structured ML objects via 'show'[55] and '(cm-(mouseright))'[312]

ForceQUIT refresh_mlob_def_cache_menu {92}

ML Def not found among cached ML objects.
Perhaps it is a primitive.

Do you want to refresh the ML object definition cache?
YES.

You might also check the standard files.

GrepLastShow

MLfiles

Not done yet. doc_for_ops_as_macros {93}

(Explain how Nuprl operators are used not only as mathematical and programming notations, but also as macros in various system uses.)

Example of method described in VIEW→ doc_for_proof_refinement {26}.

Example: Copy the following term into the tactic slot of the proof below: Analyze5

To do this copying, place the point on the TACTIC slot with '(mouseleft)'[324], then place the point on the above term, then use '(c-c)'[94].

Then, assuming the point is somewhere within the proof term, use '(c-z)'[290]. This should "analyze" the 5th hypothesis into its two conjuncts, adding them as new assumptions in the subgoal.

Now, delete the tactic code and copy this into the slot: BackThru: 4

Use '(c-z)'[290] notice that now there are two subgoals that result from "backchaining" through hpothesis 4, namely the Propositions, A and B, that hyp 4 says are sufficient for showing C.

Execute '(c-z)'[290] again and note that Auto knocks off these subgoals, since they are explicitly assumed at 6 and 7.

```
top 1 1 1 1

1" A : Prop
2" B : Prop
3" C : Prop
4" A ⇒ B ⇒ C
5" A & B
6. A
7. B
⊢" C  by  <TACTIC>

<subgoals>
```

QUIT

Parenthesis Generation. doc_for_parentheses {95}

When a term is displayed, whether a subterm is parenthesized (and what parenthesization consists of) is determined by the partial order of "precedence" between individual display forms, together with some stipulations given for the parent and child display forms. It is key that precedence is a relation not between operators, and not between DISP objects, but rather between individual display forms.

The precedence relation is determined by data taken from the display form attributes sequences, and from the LAT objects of the library. (LAT is a misnomer used because the original precedence relation was a lattice.)

For explanation of precedence, VIEW→ doc_for_precedence {97}.

The determination of whether to parenthesize is made not only on the basis of the precedence relation, but also on whether the inner term display is an "open" or a "closed" form, and on further stipulations given in the outer display for when to parenthesize its children.

Avoiding Parenthesization.
A "closed" display form will never be parenthesized; a display form is declared "open" by including *Open form* among its attributes. (Using ‘(backspace)’[99] on the display spec will toggle whether *Open form* is included.)
A display form can be declared "insensitive" to a child, in which case the child will simply not be parenthesized. Such insensitivity is indicated by using * in the attributes field of the child format <[?]:[prompt]:<child attributes>>, which will then be displayed as <[?]:[prompt]>.

If the parent is sensitive to a child, and the child display form is open and unrelated to the parent form by the partial order of precedence, then the child WILL be parenthesized.

Parenthesizing precedence-related children: VIEW→ doc_for_parentheses2 {96} doc_for_parentheses2 {96}

Parenthesizing a child according to its precedence. VIEW→ doc_for_parentheses {95}

Assume the parent is sensitive to this child, and that the child is an open form.

Whether parentheses will be inserted depends upon the precedence between the child display forms and parent (or rarely, some other form), and upon the parenthesization attribute specified in the child format command.

(<self)	parenthesize the child if it has lower precedence than the parent.
(≤self)	parenthesize the child if it has precedence the same as or lower than the parent.
(<[ob].[name])	parenthesize the child if it has lower precedence than the indicated display form.
(≤[ob].[name])	parenthesize the child if it has precedence the same as or lower than the indicated display form.

(Also, each of these will parenthesize an open child of unrelated precedence.)

The latter two forms are not used much, since it's usually the precedence between the two display forms in question that is relevant, rather than some other form.
For how to name display forms with these forms, see VIEW→ doc_for_precedence {97}.

It is not necessary to use "(" and ")" as the parens; the choice of open- and close-parens text is specific to the display child format command. Deleting these characters from (<self) gives [left par]<self[right par].

Example: Here is the typical form for left-associativity.

```
Model: <model>
Attrs: *Open form*

  <[?]:[prompt]:(<self)><Formats><[?]:[prompt]:(≤self)>
-----------------------
```

It would be an open form, and would tolerate itself as a left-child without parenthesizing, but would parenthesize itself as a right-child.

The precedence relation is a partial ordering between individual display forms, which is used in generating parentheses during display (→doc_for_parentheses {95}). It is derived from the LAT objects in the library, and possibly from certain attributes attached to the display form specs themselves.

Not every partial order is a possible precedence relation; in order to define the acceptable partial orderings, we will introduce the notion of precedence expressions, and will later say how precedence is actually determined in terms of such expressions.

A display form name "OBname.n" refers to the n-th display form spec in the DISP object named OBname.

A sequence of such display form names, with <df>=<df> as the sequencing operator, is a precedence expression meaning that all the named display forms have the same precedence.

A pair (e1 < e2) of precedence expressions is a precedence expression if no display form name occurs twice, and it means that all the display forms named in e1 have lower precedence than all those of e2.

A pair (e1 e2) of precedence expressions is a precedence expression if no display form name occurs twice, and it means that all the display forms named in e1 are UNrelated by precedence to all those of e2.

The possible precedence relations are those denoted by precedence expressions. But it would not be practical to require the user to explicitly create a single precedence expression in order to define the precedence relation between the large and variable set of display forms. So, some devices have been introduced for distributing the specification of precedence throughout the collection of DISP and LAT objects.

To allow distribution of the specification for precedence between display forms, precedence expressions have been extended to allow one LAT object to refer to another, instead of having to directly use the precedence expression contained in the object so referred to. Such references are instances of "*[object]*" with the LAT object name as the string.

The remaining devices will facilitate making more robust and distributed specifications of "precedence classes", by which we mean the equivalence classes of the relation of having the same precedence.

For how precedence classes are determined VIEW→ doc_for_precedence_classes {98}.

For how precedence is determined from the LAT objects and the precedence classes (this may be obvious now) VIEW→ doc_for_precedence_derivation {99}.

doc_for_precedence_classes {98}

How precedence classes are determined (VIEW→ doc_for_precedence {97}).

We extend display form names to include the form "OBname.foo" where "foo" is a string other than a decimal numeral. By including "Name:foo" as an attribute of a display form spec, the user introduces an alias for that display form. This makes the precedence system less sensitive to the order of specs in a display object.
But when the string is empty, the name, e.g. "OBname.", is just an abbreviation for "OBname.1", the first form in the object.

We also introduce "precedence class names" which are like display form names, except the object name position is filled by the empty string, e.g., ".foo". The string should be chosen as a convenient mnemonic for a precedence class the user has in mind. We'll see how these are used below.

It will simplify our explanation of precedence classes if we introduce a similar relation on the display names together with the variations introduced above. Let's call this relation "precedence", too. Then, we shall say two display forms have the same precedence when their standard display names (of form "OBname.n") have the same precedence.

How precedence classes are determined. Below we'll use

"OBname"	to range over possible object names,
"n"	to range over decimal numerals,
"foo"	to range over non-empty strings other than decimal numerals, and
"X" and "Y"	to range over all strings.

Sameness of precedence is the strongest equivalence relation "<df>=<df>" such that:

OBname.=OBname.1	
OBname.foo=OBname.n	if the n-th display spec of the DISP OBname has the attribute Name:foo
OBname.X=OBname.n	if the n-th display spec of the DISP OBname has the attribute =X
OBname.1=OBname.n	if the n-th display spec of the DISP OBname has the attribute =FIRST
X.Y=OBname.n	if the n-th display spec of the DISP OBname has the attribute =X.Y
X1.Y1=X2.Y2	if X1.Y1 and X2.Y2 occur in the same <df>=<df> sequence within some LAT object

doc_for_precedence_derivation {99}

How precedence is determined from the LAT objects and the precedence classes.

This may now be obvious from →doc_for_precedence {97} and →doc_for_precedence_classes {98}.

So finally, the precedence relation is the one denoted by a precedence expression derived as follows:

The "top" LAT objects are those that no other LAT objects refer to with a *[object]*. Build a sequence, using (<precs> <precs>) as the sequencing operator, of all the top LAT objects, then repeatedly replace *[object]* instances by the precedence expressions in the objects they refer to.

A "class expression" is a precedence expression built using display addresses and the <df>=<df> operator. A class expression may be said to refer not only to those display addresses mentioned explicitly in it, but also to all those in the same precedence class. Replace the maximal class expressions in our precedence expression by class expressions that explicitly mention all and only those display forms in the precedence class referred to by the expression. (This is "normalizing" the form of class expressions.)

Finally, find normal form class expressions for all the classes not mentioned in our precedence expression so far, and add them to our expression by sequencing again with (<precs> <precs>).

The resulting precedence expression is of the simple kind described earlier when explaining which partial orders are possible precedence relations.