Chapter 5

Editing Terms

Terms in NUPRL are a general-purpose, uniform data-structure that serves two different purposes.

- Terms are used to represent NUPRL's *object logic*, that is the expressions and types of its type theory together with user-defined extensions as well as all NUPRL propositions. Sometimes we refer to terms of this kind as *object-language terms*.
- Nearly all library objects, that is proofs, abstractions, display forms, and even the descriptions of the NUPRL editors are represented as terms to which we refer as *system-language terms*.

Terms are either *primitive* or *abstract*. Primitive terms have fixed pre-defined meanings and are used to describe the primitives of NUPRL's type theory as well as the foundation of the NUPRL system. Abstract terms or *abstractions* (see Chapter 7.1) are defined as being equal to other, more primitive terms. The definitions of abstractions are stored in abstraction objects of NUPRL's library (see Section 4.1.1). The visual appearance of a term is governed by its *display forms* (see Chapter 7.2), which are defined in the display-form objects of NUPRL's library.

This chapter describes the internal structure of terms, the interplay between NUPRL's structured editor and display forms, and the commands for interactively viewing and editing terms in NUPRL.

5.1 Uniform Term Structure

NUPRL terms have the form $\texttt{opid}\{p_1{:}F_1,..,p_k{:}F_k\}(x_1^1,..,x_{m_1}^1.t_1;\ \ldots;\ x_1^n,..,x_{m_n}^n.t_n)$, where $\texttt{opid}$ is the *operator identifier*. The $p_i{:}F_i$ are *parameters* and the $x_1^i,..,x_{m_i}^i.t_i$ are the *bound subterms*, expressing that the variables $x_1^i,..,x_{m_i}^i$ become bound in the term t_i. The tuple $(m_1, \ldots, m_n)$, where $m_j{\geq}0$ is the *arity* of the term. Appendix A.1 describes the current parameter types and the acceptable strings for opids, parameters, and variables. The primitive operators of NUPRL's type theory are listed in Table A.1 on page 148.

When writing terms, we sometimes omit the {} brackets around the parameter list if it is empty. Note that parameters are separated by commas while subterms are separated by semicolons.

Terms are implemented as tree data structures in NUPRL's meta-language ML (see Appendices B). Most users will rarely have to work with terms on the ML level but use the term editor to view and edit terms.

5.2 Structured Editing

In mathematics one distinguishes between logical structure of objects, and the notation in which they are presented. When we talk of the *logical structure* of a term, we are thinking of the abstract mathematical object that it represents. NUPRL's *uniform term syntax*, described in the previous section, is meant to reflect the regularity of these objects.

In contrast to that, *notation* aims at a visual presentation of abstract objects on the printed page or on the computer screen. Familiar notation for mathematical expressions helps human readers to construct the described abstract object in their minds, but should not be confused with the abstract structure itself.

In mathematics notation is crucial issue. Many mathematical developments depend heavily on the adoption of some clear notation, which makes mathematics much easier to read. However, mathematical notation can be rather complex and ambiguous if one is not aware of the immediate context it is presented in. Juxtaposition of symbols, for instance, can mean function application in one place and multiplication in another.

Notation understandable by machines, on the other hand, is often restricted to source texts in ASCII files that can be easily be parsed by a computer. Programming languages allow *overloading* of operators and *implicit coercions* and resolve these ambiguities by type-checking and similar methods. For interactive theorem proving, however, parseable ASCII notation is far from being sufficient, as it is not anywhere as easy to read as mathematics in books and papers.

The NUPRL system provides an interactive editing mechanism that presents terms in mathematical notation and groups the notation in chunks that correspond to parts of the internal tree structures. Users edit the tree structure directly, so there is no need for a parser.[1] Such editors are often called *structured editors*. The advantages of structured editors are:

- Structured editors allow using a notation that is ambiguous to a machine but unambiguous to a human who is aware of its full context.
- Formal notation is not limited to ASCII characters. NUPRL uses a single 8-bit font of up to 256 characters for displaying formal text on the screen while it is being edited and provides mechanisms for integrating LaTeX and Display PostScript-like technology to generate almost text-book quality displays. Currently the latter is only used for printing formal mathematical text but not for editing it.
- Formal notation may become context dependent. Theorems, definitions and proofs may contain local abbreviations and implicit information.
- Notation can be freely changed without altering the underlying logical structure of terms.
- Structured editors link abstract terms to notation. Users who find a particular notation confusing only need to point and click the mouse on the notation in question in order to receive a formal definition and possibly additional explanations.

The main disadvantage of structured editors is that they are quite different from conventional text editors like Emacs. Users have to get used to entering and editing terms in accordance with their tree structure instead of their textual appearance. They also have less control over formatting, since all display formatting is done automatically and may only be influenced by display forms (see Chapter 7.2). It may take a while to learn how to use a structured editor and to take advantage of its capabilities. Suggestions for improvements from users of the NUPRL editor are always welcome.

5.2.1 Term Display

To associate a chunk of notation with a term, NUPRL's term editor relies on *display form definitions*. Binary addition, for instance, is represented by the term `add(`x`;`y`)`, but conventionally written with

[1] An alternative approach, taken for instance by MATHEMATICA [Wol88], is to use a rich parseable ASCII syntax for input and then to process the input with pretty-printing routines for formatted output like Display PostScript.

the symbol + in infix notation. We could write the notation chunk as $\boxed{\square\ \texttt{+}\ \square}$, where the $\square$'s are holes for the two subterms, and the outer box shows the extent of the chunk. We call these holes *term slots*, because they can be filled by terms. In a display form definition we usually label term slots with variables to indicate how term slots correspond to the logical structure of a term. The display form for addition, for instance, can be defined as

$$\boxed{\boxed{x}\ \texttt{+}\ \boxed{y}} \quad \equiv_{\text{df}} \quad \texttt{add}(x\texttt{;}\ y)$$

where $a \ =_{\text{dform}}\ b$ means that a is defined as the *display form* for b. Term slots stretch to accommodate the terms inserted in them. For instance, the term `add(mul(1;2);3)` will be shown as

$$\boxed{\boxed{\boxed{\texttt{1}}\ \texttt{*}\ \boxed{\texttt{2}}}\ \texttt{+}\ \boxed{\texttt{3}}}$$

NUPRL automatically adds parentheses according to display form *precedences*. When a display form of lower precedence is inserted into the slot of display form with higher precedence, parentheses are automatically inserted to delimit the slot. For instance, if we assign the display form for mul(x;y a higher precedence than the one for add(x;y) then the term `add(mul(1;2);3)` is displayed without parentheses, while `mul(1;add(2;3))` is displayed as

$$\boxed{\boxed{\texttt{1}}\ \texttt{*}\ \boxed{(\boxed{\texttt{2}}\ \texttt{+}\ \boxed{\texttt{3}})}}$$

Note that parentheses are inserted merely to disambiguate the notation for a human reader, who cannot see the term slots but only ⌞`1 * (2 + 3)`⌟.

In addition to term slots and the fixed components of a notation chunk, some display forms contain *text slots*. These are slots in a definition that are filled with text strings like values for term parameters and names of binding variables. A universally quantified formula, for instance, is represented by the term ⌞all(T;x.P)⌟, meaning that "for all x of type T, the proposition P is true" (note that free occurrences of the variable x become bound in P). To generate the usual notation ⌞$\forall x$:T. P⌟ for this formula, we use the display form definition

$$\boxed{\forall\ ⌜x⌟ : \boxed{T}.\ \boxed{P}} \quad \equiv_{\text{df}} \quad \texttt{all}(T\texttt{;}\ x\texttt{.}P)$$

where ⌜ ⌟ is used to indicate a text slot.

Two display form definitions in NUPRL are rather special: the display form definition for variable terms, and the display form definition for natural numbers. Both display forms have only a single text slot, and no other printing or whitespace characters.

$$\boxed{⌜x⌟} \quad \equiv_{\text{df}} \quad \texttt{variable}\{x\texttt{:v}\}$$

$$\boxed{⌜i⌟} \quad \equiv_{\text{df}} \quad \texttt{natural\_number}\{i\texttt{:n}\}$$

In general, a display form for a term is made up of text and term slots, interspersed with printing and space characters. We can annotate display forms with *formatting commands* that specify where line breaks can be inserted, and how to control indentation. The structure and appearance of display form definitions, which are contained in *display* objects in NUPRL theories, as well as the effect of precedences on parenthesization is described in detail in Chapter 7.2.

NUPRL's term editor builds the notation for a term from the display forms associated with each node of the term tree. Thus the structure of the notation mirrors the tree structure of the term. The *display form tree* of a term is the tree structure that one actually edits with NUPRL's term editor. In a display form tree, each display form and each slot is considered a node of the tree. If a slot is not empty, it is identified with the display form tree or the text string filling it. All the slots of a display form are considered to be the immediate children of the display form and the editor considers slots ordered in the order they appear, left to right, in display form definitions.

In this manual, we refer to terms by their display notation rather than their abstract syntax, unless we want to emphasize their logical structure. Also, in our description of the editor, we talk informally about nodes of terms, when we are actually referring to nodes of the corresponding display-form trees.

5.2.2 Editor Modes

Users can navigate through a term by moving a cursor (sometimes called the *point*). Depending on the position of the cursor, the term editor will be in one of three modes, which are indicated by the shape of the cursor.

In *term mode* the cursor is positioned at some node of the term tree. The term node is indicated by highlighting its notation and the notation for all its subtrees. The highlighting is usually achieved by using reverse video; swapping foreground and background colors. For example,

∀i:ℤ. ∃j:ℤ. (j = (i + 1))

indicates that a *term cursor* is at the subterm ⌞`j = i + 1`⌟. Occasionally a term has no width, and a term cursor on such a term is displayed as a thin vertical line. In this document, we indicate such a cursor by ‖. In term mode, keystrokes corresponding to printing characters form (parts of) editor commands.

In *text mode* the cursor is positioned within a text slot and displayed as ⌞|⌟. Keystrokes corresponding to printing characters cause those characters to be inserted at the position of the cursor. The *text cursor* is significantly thinner than the term cursor on a no-width term, so it should be easy to distinguish the two. It may be positioned either between two adjacent characters, before the first character of a text slot, or after the last. Valid text cursors for the text string ⌞`abcdef`⌟, for instance include

|abcdef   abc|def   abcdef|

There is a potential ambiguity as to *which* text slot a text cursor is at. Consider, for instance, two adjacent text slots containing the strings ⌞`aaa`⌟ and ⌞`zzz`⌟ and the following text cursor:

aaa|zzz

Display forms are designed that this kind of situation should never occur.

Certain cursor motion commands are designed for moving around a term's display character-by-character as with a conventional text editor. In this case the cursor occupies a single character position on the screen. If possible, the editor uses a text cursor. Otherwise it uses a *screen cursor*, which is displayed by outlining the character. For instance, if we had the following text cursor in a term:

∀|i:ℤ. ∃j:ℤ. (j = (i + 1))

then a 'move-left-one-character' command would leave a screen cursor over the character ⌞∀⌟.

∀i:ℤ. ∃j:ℤ. (j = (i + 1))

In screen mode, keystrokes corresponding to printing characters form parts of editor commands. In the rest of this document we will never have to explicitly represent a screen cursor, so all outlined terms should be interpreted as term cursors.

5.2.3 Term and Text Sequences

The term editor has special features for handling certain kinds of sequences of terms, which makes them appear much like terms with variable numbers of subterms. A *term sequence* is constructed by iteratively pairing term slots in a right-associative way and displayed as a linear sequence. A sequence containing 4 empty term slots, for instance, might be displayed as:

```
(□, □, □, □)
```

Different kinds of term sequences have different pairing terms, a special term to represent the empty sequence, different left and right delimiters, (the ⌞(⌟ and ⌞)⌟ respectively in the example), and different element separators (the ⌞,⌟ in the example). Delimiters and separators in term sequences always consist of at least one character.

The editor considers all the term slots of the sequence as siblings in the display form tree, and the whole sequence as their immediate parent. Often, the editor does not distinguish between a term and a one-element sequence containing that term but treats a term as a one element sequence. Thus for nearly all purposes the internal structure of sequences can safely be ignored.

A *text sequence* is a text string in which characters may be replaced by terms. Text sequences are primarily used for proof tactics and other ML code, for comments, and for the left-hand sides of display forms. The editor presents a text sequence as a display form with alternating text and term slots. In contrast to term sequences, text sequences normally have no delimiters or element separators. They are however easily identified because they usually occur in well-defined contexts. An example of a text sequence is the ML expression:

```
With 'n + 1' (D 0)◊
THENW TypeCheck◊
```

This text sequence consists of 3 term slots filled with the terms ⌞'n + 1'⌟, ⌞◊⌟, and ⌞◊⌟, and 4 text slots filled with the text strings ⌞With ⌟, ⌞ (D 0)⌟, ⌞THENW TypeCheck⌟, and ⌞⌟ (the null or empty text string). The ⌞◊⌟'s are new-line terms, which are usually kept invisible and only shown with a printing character for illustration purposes. New-line characters in text should be avoided as much as possible, as this simplifies the display formatting algorithm.

5.3 Term Editor Windows

Term editor windows are used for viewing and editing terms. Except for the navigator and proof editor windows, most windows opened by NUPRL are term editor windows. Each window displays a single display form tree representing a single term. All editing operations can be carried out from the keyboard alone, but the editor accepts input from the keypad and the mouse as well.

Input characters typed at the keyboard in multi-character commands are echoed as highlighted text near the position of the cursor, and can be corrected by using DEL. Certain key combinations are bound to editor commands, which also may be invoked explicitly by typing ⌞⟨C-M-x⟩*command-name*⌟. The default key bindings are intended to be reminiscent of Emacs's key bindings. A summary of all the key bindings that we will describe below can be found in Table 5.8 at the end of this chapter. Users who wish to use alternative key bindings may customize the term editor as described in Section 5.8.

The editor adjusts the display of an object in a window to the size of the window. If the window is too small, not all the object can be displayed at once. In this event, one can resize the window, or scroll the window up and down. Sometimes, if the window is to narrow, some subterms are *elided*.

The display form tree for an elided subterm is replaced by ⌞...⌟. Currently, the only way to to un-elide a subterm is to widen the window as much as possible. Eventually, one will be able to examine elided subterms by moving the root display form of an editor window to some term tree position other than the term root.

Term editor windows are opened when a user access a library object through the navigator. Opening a proof object opens a *proof editor window*, which in turn opens a term slot for entering the goal sequence and text slots for entering refinement rules (see Chapter 6 for details). To close and change term editor windows, one may use the following commands and key sequences.

⟨C-Z⟩	EXIT	save, check, and close window
⟨C-Q⟩	QUIT	close window without saving
⟨C-J⟩	JUMP-NEXT-WINDOW	jump to next window

EXIT first saves a copy of the object. It then checks the object before closing the window. This *checking* has the same effect on library objects as using the ML `check` command. If the check fails, then the window is left open. If you still want to close the window, use QUIT instead. Separate save and check commands are described in Section 5.7.

QUIT is an abort command. It closes the window, abandoning any changes made to the window since it was last checked by attempting EXIT.

JUMP-NEXT-WINDOW allows one to cycle around all the currently open windows, including any proof editor windows.

5.4 Entering Information

Term editor windows for ML and comment objects initially open in text mode while abstractions and display forms usually open in term mode.[2] Special display forms allow opening text slots in term windows while special key sequences are used to open term slots within text sequences (see Section 5.4.2 below).

5.4.1 Inserting Text

The following commands are for inserting text whenever the editor is in text mode.

x	INSERT-CHAR-x	insert char x
⟨C-#⟩*num*	INSERT-SPEC-CHAR-*num*	insert special char x
↵	INSERT-NEWLINE	insert newline

Standard ASCII printing characters (including space) self insert in text mode. Non-standard characters can be inserted using INSERT-SPEC-CHAR-*num*, where *num* is the decimal code for the character. Table 5.1 lists the currently available special character codes.[3] Alternatively, special characters can be copied from the object `FontTest` in the library theory `core-1`.

The INSERT-NEWLINE command is only appropriate in text sequences. Within terms the newline character is actually a term whose display is controlled by the display layout algorithm.

[2] Currently, newly created objects contain an empty term slot. Removing this slot in ML and comment objects with ⟨C-C⟩ puts the editor into text mode. The term slot in abstractions and display forms cannot be removed.

[3] The NUPRL fonts may be extended in the future. Executing the UNIX command `xfd -font nuprl-13 &` pops up a display of the actual standard NUPRL font. Clicking LEFT on a character results in its decimal code being displayed.

	0	1	2	3	4	5	6	7	8	9
120									$\mathbb{P}$	$\mathbb{R}$
130	$\mathbb{N}$	$\mathbb{C}$	$\mathbb{Q}$	$\mathbb{Z}$	$\mathbb{U}$	$\Leftarrow$		$\Rightarrow$	$\lceil$	$\rceil$
140	$\vdash$	$\int$	$\cdot$	$\downarrow$	α	β	$\wedge$	$\neg$	$\in$	π
150	λ	γ	δ	$\uparrow$	$\pm$	$\oplus$	∞	∂	$\subset$	$\supset$
160	$\cap$	$\cup$	$\forall$	$\exists$	$\otimes$	$\leftrightarrow$	$\leftarrow$		$\neq$	$\diamond$
170	$\leq$	$\geq$	$\equiv$	$\vee$		$\_$	$\rightarrow$	Σ	Δ	Π
180	$\times$	$\div$	$^{+}$	$^{-}$	^{0}	$\subseteq$	$\supseteq$	$_{0}$	$_{1}$	$_{2}$
190	$_{3}$	$\circ$	$\mathbb{B}$	ρ	$_{a}$	$_{q}$	$_{b}$	$_{d}$	$_{c}$	$\vert$
200					^{1}	^{2}	^{3}	μ	ϵ	θ
210	$\bigcap$	$\bigcup$	$\hat{=}$	$\doteq$	$\Leftrightarrow$	$\not\rightarrow$	$\sqsubseteq$		$\top$	$\bot$
220			$\mapsto$		ä	ö	ü	ß	Ä	Ö
230	Ü	T	F			$\emptyset$	$\notin$	$\not\subseteq$		$\not\sqsubseteq$
240	Γ	Λ	$\backslash$	Θ	σ				$_{i}$	$_{j}$
250	$_{k}$	$_{l}$	$_{m}$	$_{n}$						

Table 5.1: NUPRL special character codes

5.4.2 Adding and Removing Slots

⟨C-`U`⟩	OPEN-LIST-TO-LEFT	open slot to left of cursor
⟨M-`U`⟩	OPEN-LIST-TO-RIGHT	open slot to right of cursor
⟨C-`O`⟩	OPEN-LIST-LEFT-AND-INIT	open and initialize slot to left
⟨M-`O`⟩	OPEN-LIST-RIGHT-AND-INIT	open and initialize slot to right
⟨C-`C`⟩	CLOSE-LIST-TO-LEFT	close slot and move left
⟨M-`C`⟩	CLOSE-LIST-TO-RIGHT	close slot and move right

NUPRL's term editor makes it possible to combine informal, semi-formal, and formal knowledge by inserting terms into text sequences. These terms are displayed according to their display forms and surrounded by ordinary text.

To make this possible, a term slot must be opened and then initialized. The latter inserts a *term holder* into the term slot, which initially looks like ⌞`'[term]'`⌟. The commands OPEN-LIST-LEFT-AND-INIT and OPEN-LIST-RIGHT-AND-INIT combine these two commands and position the cursor at the empty slot of the term holder. Terms may now be inserted into the slot as usual (see Section 5.4.3 below).

The commands for opening, initializing, and removing slots apply both in text and in term mode and thus have a slightly more general meaning than just described.

In term mode, OPEN-LIST-TO-LEFT and OPEN-LIST-TO-RIGHT only apply if the term cursor is an element of a (term or text) sequence. They add a new empty slot to the left (or right) of the cursor and move the cursor to the new empty slot. On an empty term sequence, both commands have the same effect; they simply delete the nil sequence term. In text mode, both commands open up an empty term slot at the text cursor, and leave the cursor at the new slot.

With text or term sequences represented by a single term, these commands infer the kind of sequence to create from context. Occasionally with term sequences, more than one kind of sequence is permitted in a given context (for example, in precedence objects) and in such cases you can use explicit term insertion commands to create the sequence. Such ambiguity should not arise with text sequences.

OPEN-LIST-LEFT-AND-INIT and OPEN-LIST-RIGHT-AND-INIT are similar, but if there is some obvious term to insert in the opened up slot, then that term is automatically inserted and the cursor is left at an appropriate position in the new term.

If a term cursor is at an empty term slot in a term sequence, the commands CLOSE-LIST-TO-LEFT and CLOSE-LIST-TO-RIGHT delete the slot, and then (if possible) move the cursor to the element to the left or right respectively of the slot just deleted. If the term slot is filled with a term, that term is deleted as well. If the term slot is in a text sequence, these commands leave a text cursor at the position of the deleted slot.

5.4.3 Inserting Terms

In structured editing, one usually enters terms in a top-down fashion, starting with the root of the term tree and working on down to the leaves. This means that one has to work with *incomplete* terms. For example, at an intermediate stage of entering the term ⌞`∀i:ℤ. ∃j:ℤ. j = i + 1`⌟ one might be presented with the term:

`∀i:ℤ. ∃|[var]:[type]. [prop]`

Here `[var]`, `[type]` and `[prop]` are *place-holders* for slots in the display of the existential quantifier. If a slot has a place-holder, we say that the slot is *empty*, or *uninstantiated*. Place-holders for subterms of a term are term slots while others are text slots. The labels that appear in the place-holders (the `var`, `type` or `prop` in the example above) are controlled by the definition of the term's display form. If a text (term) slot contains a a text string (term) we say that slot is *filled* or *instantiated*. If a display form has no uninstantiated slots, then it is considered *complete*. Place-holders re-appear when the contents of slots are removed.

name	INSERT-TERM-*name*	insert *name* into empty slot
⟨C-I⟩*name*	INSERT-TERM-LEFT-*name*	insert *name*, using existing term as left subterm
⟨M-I⟩*name*	INSERT-TERM-RIGHT-*name*	insert *name*, using existing term as right subterm
⟨C-S⟩*name*	SUBSTITUTE-TERM-*name*	replace existing term with *name*
⟨C-M-I⟩	INIT-TERM	initialize term slot
⟨C-M-S⟩	SELECT-DFORM-OPTION	selects display form variations

To insert terms into term slots one may use the editor commands listed above. In these commands, *name* is a string of characters naming a new term to be inserted. The interpreter for *name* strings checks each of the following conditions until it finds one which applies.[4]

1. *name* is an editor command enabled in a particular context.
2. *name* is an *alias* for some display form, defined in in the library object for that display form.
3. *name* is the name of a display form object. In this case it refers to the first display form defined in that object.
4. *name* is of the form *ni* where *n* is the name of a display form object and *i* is a natural number. *ni* refers to the *i*-th display form definition in the object named *n*. Definitions in objects are numbered starting from 1.

[4]Note that this order gives display form names and aliases preference over abstraction names. The operator identifier of a term can *not* be used to identify a term, if it is neither of these three. This is particularly important when referring to the elementary terms of type theory. To find out how to refer to a particular term, mark the term and enter ⟨C-X⟩`df` to see its display form or ⟨C-X⟩`ab` to see the abstraction that defines it, if there is one.

5. *name* is the name of an abstraction object. In this case it refers to the earliest display form in the library for that abstraction.
6. *name* is all numerals, then the term referred to is the term ⌞`natural-number{`*name*`:n}()`⌟ term of NUPRL's object language.
7. If none of the above applies, *name* is assumed to refer to the term ⌞`variable{`*name*`:v}()`⌟.

Since names always have acceptable extensions as variable names, the editor does not interpret *name* until some explicit terminator such as SPC (NO-OP) or a cursor motion command is typed. ↵ (RIGHT-EMPTY-SLOT, see Section 5.5) is a particularly useful terminator.

INSERT-TERM-*name* is only applicable at empty term slots. It results in the display form referred to by *name* being inserted into the slot. If *name* is terminated by a NO-OP, then a term cursor is left at the new term. If *name* is terminated by some cursor motion command, then that command is obeyed.

INSERT-TERM-LEFT-*name* is intended for use at a filled term slot. Its behavior is to:

1. save the existing term in the slot, leaving the slot empty,
2. insert the new display form referred to by *name* into the slot,
3. paste the saved term into the leftmost term slot of the new display form. If the new display form has no term slots, then the saved term is lost.

INSERT-TERM-RIGHT-*name* behaves in a similar way to INSERT-TERM-LEFT except that in step 3, the saved term is pasted into the rightmost term slot of the new display form.

SUBSTITUTE-TERM-*name* replaces one display form with another that has the same sequence of child text and term slots. The children of the old display form become the children of the new one. If the new display form has a different sequence of children SUBSTITUTE-TERM-*name* tries something sensible, but in these cases it is safer to explicitly cut and paste the children.

INIT-TERM initializes a term slot to some default term. INITIALIZE-TERM is automatically invoked by NUPRL to initialize new windows. To re-initialize a window, place a term cursor at the root of the term in the window, delete the term and then give the INITIALIZE-TERM command. The default terms for particular contexts are described in various sections of this document. If no default has been designated, INITIALIZE-TERM does nothing.

SELECT-DFORM-OPTION selects an alternative display form for the term where the term cursor is positioned. For instance, if term cursor is positioned at an independent function type, it selects the more general dependent-function display form.

5.4.4 Adding New Terms

The term editor recognizes certain input sequences as indicating that a new term should be created. A new term structure can be created as follows:

- Position a term cursor at an empty slot and enter the letters of the new term's opid
- Enter a (possibly empty) list of parameter types for the new term (see Section A.1.2), abbreviated by single letters. The list has to be delimited by ⌞{⌟ and ⌞}⌟ characters, and elements must be separated by ⌞,⌟ characters. Empty lists of parameter types are optional.
- Enter a list of subterm arities, i.e. numbers designating the number of binding variables for each subterm. The list must be delimited by ⌞(⌟ and ⌞)⌟ characters, and elements must be separated by ⌞;⌟ characters. This list has to be entered even when it is empty.

Upon receiving the opid, the list of parameter types, and the list of subterm arities the editor creates a new term in uniform syntax with appropriate place-holders for parameters and subterms. For instance, entering `myid{n,t}(0;1)` creates the term

```
myid{[natural]:n, [token]:t}([term]; [binding].[term])
```

5.4.5 Exploded Terms

Exploded terms provide access to the internal structure of a term, allowing users to change its opid, the number and kind of its parameters, or its arity. They are most commonly used for creating new terms in an abstraction or for changing the definition of an abstraction.

A term constructor is *exploded* by replacing it by a special collection of terms that make it possible to edit its structure. Exploded terms may be generated from scratch by typing `exterm` into an empty term slot, or by positioning the term cursor over a term and typing ⟨C-X⟩`ex`. Exploded terms may be imploded again into the term which the exploded term represents by typing ⟨C-X⟩`im`. The commands for editing exploded terms are summarized in the table below.

⟨C-X⟩`ex`	EXPLODE-TERM	explode term at cursor
⟨C-X⟩`im`	IMPLODE-TERM	implode term at cursor
`exterm`	INSERT-TERM-`exterm`	insert new exploded term
`lparm`	INSERT-TERM-`lparm`	insert level expression parameter
`vparm`	INSERT-TERM-`vparm`	insert variable parameter
`tparm`	INSERT-TERM-`tparm`	insert token parameter
`sparm`	INSERT-TERM-`sparm`	insert string parameter
`nparm`	INSERT-TERM-`nparm`	insert natural number parameter
⟨C-O⟩	OPEN-LIST-TO-LEFT	open new slot to left
⟨M-O⟩	OPEN-LIST-TO-RIGHT	open new slot to right

The editor is somewhat intelligent when new slots with OPEN-LIST-TO-LEFT and OPEN-LIST-TO-RIGHT. Depending on the context, the new slot will be a placeholder for a bound term (`[bterm]`), bound variable (`[bvar]`), or a parameter (`[parm]`).

To show how exploded terms are used, we walk through the entry of the term ⌞`foo{bar:s}(A;x.B)`⌟. Create an empty term slot and enter ⌞`exterm`SPC⌟. The highlighted term should look like:

```
EXPLODED<<[opid] {}([bterm])>>
```

Enter the opid ⌞`foo`⌟ to get:

```
EXPLODED<<foo| {}([bterm])>>
```

Click LEFT on the `}`, and you should get a null width term cursor sitting on an empty term sequence for parameters.

```
EXPLODED<<foo {|}([bterm])>>
```

Enter ⟨C-O⟩ to add a new slot to the parameter sequence:

```
EXPLODED<<foo {[parm]}([bterm])>>
```

Insert the string parameter with text `bar` by typing ⌞`sparm`↵`bar`⌟:

```
EXPLODED<<foo {bar|:s}([bterm])>>
```

Click LEFT on the `[bterm]` and enter ⟨C-O⟩⟨C-O⟩ to make a two element sequence for bound terms, leaving the cursor on the left-most element.

`EXPLODED<<foo {bar:s}([bindings].[term];[bindings].[term];)>>`

Enter ⟨C-O⟩ ↵ to open up a slot in the sequence, and enter a binding variable term:

`EXPLODED<<foo {bar:s}(|.[term];[bvar],.[term];)>>`

You could now go ahead and fill in the binding variable, and subterm slots by typing ␣A ↵ x ↵ B ↵␣.

`EXPLODED<<foo {bar:s}(.A;x,.B;)>>`

Finally, click LEFT on any part of `EXPLODED` and then enter ␣⟨C-X⟩im␣ to implode the exploded terms. You should now have the term:

`foo{bar:s}(.A;x.B)`

In general, when imploding and exploding terms the parameter values, binding variable names, and subterms stay the same, so entering and/or editing them when a term is exploded has the same effect as when the term is imploded.

5.5 Cursor and Window Motion

Nuprl's editor supports two basic forms of cursor motion. *Screen oriented* cursor motion commands ignore the structure of the term in the window and allow one to quickly navigate to parts of the screen. In contrast to that *tree oriented* cursor motion commands follow the structure of the term tree. In addition to key sequences *mouse commands* allow easy jumping around terms and *search commands* allow moving the cursor to a particular substring.

5.5.1 Screen Oriented Motion

⟨C-P⟩	SCREEN-UP	move cursor up 1 character
⟨C-B⟩	SCREEN-LEFT	move cursor left 1 character
⟨C-F⟩	SCREEN-RIGHT	move cursor right 1 character
⟨C-N⟩	SCREEN-DOWN	move cursor down 1 character
⟨C-A⟩	SCREEN-START	move to left side of screen
⟨C-E⟩	SCREEN-END	move to right side of screen
⟨C-L⟩	SCROLL-UP	scroll window up 1 line
⟨M-L⟩	SCROLL-DOWN	scroll window down 1 line
⟨C-V⟩	PAGE-DOWN	move window down 1 page
⟨M-V⟩	PAGE-UP	move window up 1 page
⟨C-T⟩	SWITCH-TO-TERM	switch to term mode

Screen oriented cursor motion commands are listed in the table above. After a screen cursor command the cursor is always either in text mode or screen mode. To switch to term mode, one may use the SWITCH-TO-TERM command if the cursor is over the printing character of a display form. If the cursor is moved over the top or bottom of the display, the window scrolls appropriately. There are also explicit window scrolling commands.

5.5.2 Tree Oriented Motion

⟨M-P⟩	UP	move up to parent
⟨M-B⟩	LEFT	structured move left
⟨M-F⟩	RIGHT	structured move right
⟨M-N⟩	DOWN-LEFT	move to leftmost child
⟨M-M⟩	DOWN-RIGHT	move to rightmost child
⟨M-A⟩	LEFTMOST-SIBLING	move to left-most sibling
⟨M-E⟩	RIGHTMOST-SIBLING	move to right-most sibling
⟨M-<⟩	UP-TO-TOP	move up top of term
⟨C- LFD ⟩	RIGHT-LEAF	next leaf to right
⟨M- LFD ⟩	LEFT-LEAF	next leaf to left
↵	RIGHT-EMPTY-SLOT	next empty slot to right
⟨C- ↵ ⟩	RIGHT-EMPTY-SLOT	next empty slot to right
⟨M- ↵ ⟩	LEFT-EMPTY-SLOT	next empty slot to left

UP, LEFT, RIGHT, DOWN-LEFT, DOWN-RIGHT are the basic walking commands. These commands recognize text and term sequences, and skip over their internal structure. Within text slots, LEFT and RIGHT stop at each word.

RIGHT-LEAF, LEFT-LEAF, RIGHT-EMPTY-SLOT, LEFT-EMPTY-SLOT are particularly good for rapidly moving around terms, since you can often get where you want to go by just repeatedly using one of them. Note that ↵ is not bound to RIGHT-EMPTY-SLOT within text sequences. In that case, you need to use ⟨C- ↵ ⟩.

5.5.3 Mouse Commands

LEFT	MOUSE-MARK-THEN-SET-POINT	set mark then point
⟨C- LEFT ⟩	MOUSE-MARK-THEN-SET-POINT-TO-TERM	set mark then point to term

MOUSE-MARK-THEN-SET-POINT first sets the *mark*, an auxiliary cursor for marking regions (see Section 5.6.2) at the current position of the *editor cursor* and *then* sets the editor's cursor, the *point*, to where the mouse is pointing. MOUSE-SET-POINT results in a text cursor if one is valid between the character pointed to and the character to the immediate left. If there is a null width term to the immediate left of the mouse, it results in a term cursor pointing to that term. Otherwise, the editor cursor is set to the most smallest term that contains the character being pointed to. MOUSE-MARK-THEN-SET-POINT-TO-TERM is like MOUSE-MARK-THEN-SET-POINT except that point is always set to the term immediately surrounding the character being pointed to.

5.5.4 Search for Subterms

⟨M-s⟩	SET-SEARCH-MODE	initialize substring search
⟨C-s⟩	VIEW-SEARCH-FORWARDS	search forward
⟨C-r⟩	VIEW-SEARCH-BACKWARDS	search backward

SET-SEARCH-MODE initializes the search for a substring. It expects a substring to be be entered and then sets the cursor to the next text or term slot that contains this substring. Thus a user has to enter ⌞⟨M-s⟩*substring* SPC⌟ to search for the first occurrence of *substring*. Currently, *substring* cannot contain whitespace.

As long as the editor is in search mode VIEW-SEARCH-FORWARDS move the cursor to the next slot containing *substring*. VIEW-SEARCH-BACKWARDS does the same moving backwards.

5.6 Cutting and Pasting

Cut-and-paste commands work on terms, segments of text slots, and segments of text and term sequences. In this section we refer to these collectively as *items*. Items can be saved on a *save stack*, in which they are represented as terms. Often it is possible to cut one kind of item and then paste it into another kind of context. For example, one can cut a term and paste into text sequence, or cut a segment of text from a text slot and paste into a term sequence. Within the context of NUPRL's editor, cut-and-paste commands have the following meaning:

SAVE**:** push a copy of an item onto the save stack, leaving the item in place. Similar to *copy-as-kill* in Emacs.

DELETE**:** remove an item from a buffer without saving it.

CUT**:** (= SAVE + DELETE) remove an item from a buffer and push it onto the top of the *save stack*. Similar to *kill* in Emacs, although NUPRL does *not* append together items that were cut immediately one after the other.

PASTE**:** insert the item on top of the stack back into a buffer, removing it from the stack. Successive pastes thus retrieve items that were saved earlier.

PASTE-COPY**:** insert the item on top of the stack back into a buffer without removing it from the stack. This is useful for making several copies of an item. Similar to *yank* in Emacs.

PASTE-NEXT**:** remove the item just pasted from the buffer and paste the item that is now on top of the stack. Can only be used immediately after a PASTE. By repeating PASTE-NEXT one may back through the save stack for some desired item. Similar to *yank-next* in Emacs.

5.6.1 Basic Commands

DEL	DELETE-CHAR-TO-LEFT	delete char to left of text cursor
⟨C-D⟩	DELETE-CHAR-TO-RIGHT	delete char to right of text cursor
⟨M-D⟩	CUT-WORD-TO-RIGHT	cut word to right of text cursor
⟨C-K⟩	CUT	cut term
⟨M-K⟩	SAVE	save term
⟨C-M-K⟩	DELETE	delete term
⟨C-Y⟩	PASTE	paste item
⟨M-Y⟩	PASTE-NEXT	delete item then paste next item
⟨C-M-Y⟩	PASTE-COPY	paste copy of item

DELETE-CHAR-TO-LEFT and DELETE-CHAR-TO-RIGHT are conventional character deletion commands. They only remove the character without saving it on the save stack. They can be used in any text slot of a term or in a text sequence and also work on newline terms in text sequences.

CUT-WORD-TO-RIGHT cuts the word to the right of a text cursor. If a term is to the immediate right of a text cursor in a text sequence, then that term is cut. CUT, SAVE, and DELETE work on a term underneath a term cursor as described above. These commands work fine on terms in text and term sequences. Note that deleting a term leaves an empty term slot.

When a term cursor is at an empty term slot, the PASTE and PASTE-COPY commands paste the term on top of the stack into the slot. PASTE-NEXT replaces the last term pasted with the term on top of the paste stack. It should only be used immediately after a PASTE or a previous PASTE-NEXT.

5.6.2 Cutting and Pasting Regions

A *region* is a segment of any text slot, or a segment of a text or term sequence. A region is delimited by the editor's term or text cursor and an auxiliary text or term cursor position. Following Emacs's terminology, we call the cursor's position the *point* and the auxiliary cursor position the *mark*. It does not matter whether mark is to the left or the right of point when selecting a region. In what follows, we call the left-most of point and mark the *left delimiter*, and the right-most the *right delimiter*. If a term is used as a region delimiter, the term is included in the region.

Various regions are acceptable. For selecting a text string in a text slot, both delimiters must be text cursor positions. For selecting a segment of a term sequence, both delimiters must be term cursor positions. For selecting a segment of a text sequence, text or term cursor positions may be used for each delimiter. The commands for cutting and pasting regions are shown below.

⟨C- SPC ⟩	SET-MARK	set mark at point
⟨C-X⟩⟨C-X⟩	SWAP-POINT-MARK	swap point and mark
⟨C-W⟩	CUT-REGION	cut region
⟨M-W⟩	SAVE-REGION	save of region
⟨C-M-W⟩	DELETE-REGION	delete region

SET-MARK sets the mark to the current cursor position while SWAP-POINT-MARK swaps the mark and the editor cursor. This command is often used to check the mark's position.

SAVE-REGION saves a region onto the save stack. DELETE-REGION deletes the region. If the region is of a text slot or a text sequence, DELETE-REGION leaves a text cursor at the old position of the region. If the region is of a term sequence, an empty term slot is left in place of the region. CUT-REGION has the same effect as a SAVE-REGION followed by a DELETE-REGION.

The paste commands for regions are the same as the basic paste commands described above. One can paste with a text cursor in a text slot or text sequence, and a term cursor at any empty term slot. Pasting a sequence into another sequence of the same kind merges the pasted sequence into the sequence being pasted into. In this event, the point is set to be the left-delimiter for the sequence just pasted and the mark is set to be the right-delimiter. This ensures proper functionality for the PASTE-NEXT operation. Otherwise, pasting and item into a sequence always incorporates the item as a single sequence element and both the mark and point are set to that element. Note that it does not make sense to try to paste a term or a text sequence containing a term into a text slot that is not in a text sequence.

5.6.3 Mouse Commands

LEFT	MOUSE-MARK-THEN-SET-POINT	set mark then point
⟨C- LEFT ⟩	MOUSE-MARK-THEN-SET-POINT-TO-TERM	set mark then point to term
⟨C- MIDDLE ⟩	MOUSE-PASTE	paste region
⟨M- MIDDLE ⟩	MOUSE-PASTE-NEXT	replace last paste with new paste
⟨C-M- MIDDLE ⟩	MOUSE-PASTE-COPY	paste copy of region on stack
⟨C- RIGHT ⟩	MOUSE-CUT	cut term or region
⟨M- RIGHT ⟩	MOUSE-SAVE	save term or region
⟨C-M- RIGHT ⟩	MOUSE-DELETE	delete term or region

MOUSE-SET-POINT and MOUSE-SET-TERM-POINT first set the mark at the current editor cursor position and then set the point to where the mouse is pointing, as described in Section 5.5.3. These commands are set up so that a region can be selected by using them at both ends; after the second MOUSE-SET-POINT the mark will be at one end of the region and point will be at the other.

MOUSE-PASTE, MOUSE-PASTE-NEXT, and MOUSE-PASTE-COPY are the same as PASTE, PASTE-NEXT, and PASTE-COPY. MOUSE-CUT is the same as CUT-REGION in text sequences and text slots and the same as CUT otherwise. MOUSE-SAVE and MOUSE-DELETE behave similarly. Note all that these commands do *not* move the point before cutting and pasting.

5.7 Utilities

NUPRL's term editor provides various utility commands that are shown in the table below. The IDENTIFY-TERM, SUPPRESS-DFORM and UNSUPPRESS-DFORM commands assist in interpreting unfamiliar or ambiguous display forms. Exploding a term reveals its internal structure. Viewing abstraction and display form definitions of a term help understanding the formalization of a user-defined concept and its notation. The latter two commands can also issued via mouse commands.

⟨C-X⟩id	IDENTIFY	gives info on term at cursor
⟨C-X⟩su	SUPPRESS	suppress display form at cursor
⟨C-X⟩un	UNSUPPRESS	unsuppress display form at cursor
⟨C-X⟩ns	TERM-INSERT-NULL	insert empty string in text slot
⟨C-X⟩df	VIEW-DISP	view display form def for term
⟨C-X⟩ab	VIEW-ABS	view abstraction def of term
MIDDLE	VIEW-DISP	view display form of term
RIGHT	VIEW-ABS	view abstraction definition of term

IDENTIFY will print out in the ML Top-Loop window information on the term and display form at the current cursor position.

SUPPRESS suppresses use of the display form of all occurrences of the term pointed to by the cursor in the currently viewed object. If multiple display forms are defined for a term, a single SUPPRESS-DFORM might result in some other more general display form being selected. In this case one can repeat SUPPRESS-DFORM. When all appropriate display forms for a term are suppressed, the term is displayed in uniform syntax.

UNSUPPRESS restores a suppressed display form if the editor cursor is at a term to which that suppressed display form belongs. Display forms remain suppressed until explicitly unsuppressed or until the editor window is closed.

TERM-INSERT-NULL is useful for inserting empty text strings into text slots. Normally, when all the characters in a text slot that is outside of a text sequence are deleted, a text slot placeholder is left to indicate what kind of item should be inserted into the slot. Use this command if an empty string is what is really wanted.

VIEW-DFORM and MOUSE-VIEW-DISP open the display form object that defines the display form of the term pointed to by the editor cursor (or the mouse).

VIEW-ABSTRACTION and MOUSE-VIEW-AB open the abstraction object that defines the type theoretical meaning of the term pointed to by the editor cursor (or the mouse).

(c-U)	open term slot	DEL	delete char to left of text cursor
(cm-I)	init term slot with prl term	⟨C-D⟩	delete char to right of text cursor
(c-O)	open term slot and init	⟨M-D⟩	cut word to right of text cursor
⟨C-Q⟩	close window without saving	⟨C-K⟩	cut term
⟨C-Z⟩	save, check, and close window	⟨M-K⟩	save term
⟨C-J⟩	jump to next window	⟨C-M-K⟩	delete term
↹	jump to ML top loop	⟨C-Y⟩	paste item
x	insert char *x*	⟨M-Y⟩	delete item then paste next item
⟨C-#⟩*num*	insert special char *x*	⟨C-M-Y⟩	paste copy of item
↵	insert newline	⟨C- SPC ⟩	set mark at point
name	insert *name*	⟨C-X⟩⟨C-X⟩	swap point and mark
⟨C-I⟩*name*	insert *name*	⟨C-W⟩	cut region
⟨M-I⟩*name*	insert *name*	⟨M-W⟩	save of region
⟨C-S⟩*name*	replace with *name*	⟨C-M-W⟩	delete region
⟨C-M-I⟩	initialize term slot	⟨C-(Y⟩	paste region
⟨C-M-S⟩	selects dform variations	⟨M-Y⟩	replace last paste with new paste
⟨C-U⟩	open slot to left of cursor	⟨C-M-Y⟩	paste copy of region on save-stack top
⟨M-U⟩	open slot to right of cursor	LEFT	set mark then point
⟨C-O⟩	open slot to left and init	⟨C- LEFT ⟩	set mark then point to term
⟨M-O⟩	open slot to right and init	MIDDLE	view display form of term
⟨C-C⟩	close slot and move left	⟨C- MIDDLE ⟩	as PASTE
⟨M-C⟩	close slot and move right	⟨M- MIDDLE ⟩	as PASTE-NEXT
⟨C-P⟩	move cursor up 1 character	⟨C-M- MIDDLE ⟩	as PASTE-COPY
⟨C-N⟩	move cursor down 1 character	RIGHT	view abstraction definition of term
⟨C-B⟩	move cursor left 1 character	⟨C- RIGHT ⟩	cut term or region
⟨C-F⟩	move cursor right 1 character	⟨M- RIGHT ⟩	save term or region
⟨C-A⟩	move to left side of screen	⟨C-M- RIGHT ⟩	delete term or region
⟨C-E⟩	move to right side of screen	⟨C-X⟩id	gives info on term at cursor
⟨C-L⟩	scroll window up 1 line	⟨C-X⟩su	suppress display form at cursor
⟨M-L⟩	scroll window down 1 line	⟨C-X⟩un	unsuppress display form at cursor
⟨C-V⟩	move window down 1 page	⟨C-X⟩ex	explode term at cursor
⟨M-V⟩	move window up 1 page	⟨C-X⟩im	implode term at cursor
⟨C-T⟩	switch to term mode	⟨C-X⟩ch	check object
⟨M-P⟩	move up to parent	⟨C-X⟩sa	save object
⟨M-B⟩	structured move left	⟨C-X⟩ab	view abstraction def of term
⟨M-F⟩	structured move right	⟨C-X⟩df	view display form def for term
⟨M-N⟩	move to leftmost child	⟨C-X⟩ns	insert empty string in text slot
⟨M-M⟩	move to rightmost child	`exterm`	insert new exploded term
⟨M-A⟩	move to left-most sibling	`lparm`	insert level exp parm
⟨M-E⟩	move to right-most sibling	`vparm`	insert variable parm
⟨M-<⟩	move up top of term	`tparm`	insert token parm
⟨C- LFD ⟩	next leaf to right	`sparm`	insert string parm
⟨M- LFD ⟩	next leaf to left	`nparm`	insert natural number parm
↵	next empty slot to right	⟨C-O⟩	open bterm / parm / bvar slot to left
⟨C- ↵ ⟩	next empty slot to right	⟨M-O⟩	open bterm / parm / bvar slot to right
⟨M- ↵ ⟩	next empty slot to left		

Table 5.2: All key and mouse command

```
% ----------------------------------------------------------------------
% Arrow keys
%
% Default :
%
(UP)==(m-p)
(LEFT)==(m-b)
(RIGHT)==(m-f)
(DOWN)==(m-n)
%
% Text :
%
(RIGHT)==(-text)(m-X)screen-right
(LEFT)==(-text)(m-X)screen-left
% ----------------------------------------------------------------------
% Mouse keys:
%
(MouseLeft)==(cm-X)mouse-mark-then-set-point
(c-(MouseLeft))==(cm-X)mouse-mark-then-set-point-to-term
%
(MouseMiddle)==(m-X)view-disp
(c-(MouseMiddle))==(cm-X)mouse-paste
(m-(MouseMiddle))==(cm-X)mouse-paste-next
(cm-(MouseMiddle))==(cm-X)mouse-paste-copy
%
(MouseRight)==(m-X)view-abs
(c-(MouseRight))==(cm-X)mouse-cut
(m-(MouseRight))==(cm-X)mouse-save(m-X)blink
(cm-(MouseRight))==(cm-X)mouse-delete
%
% ----------------------------------------------------------------------
```

Table 5.3: Term-editor fragment of the standard `mykeys.macro` file

5.8 Customizing the Editor

The current key bindings of NUPRL's term editor, which are summarized in Table 5.8, are intended to be reminiscent of Emacs's key bindings and compatible with previous releases of NUPRL.

Users who wish to change the default bindings may do so by putting a file called `mykeys.macro` into their home directory. In this file one may define bindings of keys or key combinations to commands of NUPRL's term editor (as well as to navigator and proof editor commands). Bindings can be made globally or depending on the context of cursor. Table 5.3 lists the term-editor fragment of a standard `mykeys.macro` file with the default bindings. Users should exercise caution when changing global bindings, as these may have unwanted effects on the navigator and the proof editor.