The Editing and Transformation manual

June 8, 1994

Centaur	version 1.3
Documentation revision	version 1.14.3.1 94/06/08 14:53:17 vmp Merged

Contents

1 Introduction

CENTAUR offers a wide variety of structured and textual editing techniques, described in this document.

1.1 Structured editing

Structured editing is based on an abstract syntax specification. The generic *ctedit* editor (described fully in the Ctedit manual) displays an abstract syntax tree and allows it to be modified interactively. Structured editing ensures syntactic correctness; only complete syntactic elements can be removed from or added to a program. Two structured editing techniques are available:

- *Generic editing* : Certain generic basic editing strokes —copy, cut, paste, insert, and undo— may be performed for any abstract syntax tree. These operations are made available through a graphical interface (the **Edit** pulldown of a ctview or the calculator box made visible from the **Editing Tools** pulldown).

- *Menu mode* : The menu mode consists of a number of graphic tools that together facilitate structural development and program transformation. Menu mode enables the rapid construction of a program by selecting and combining *templates*. A template is a fragment of abstract syntax interspersed with metavariables. The current selection in the main program establishes a context to which menus react, displaying the appropriate templates for that context. Contexts are usually syntactic, but may also be defined according to semantic concerns.

  The primary application of menu mode is the *guided editing mode*. In this mode, clicking on an applicable template replaces the currently selected fragment with the menu entry. Menu entries are generated from abstract syntax specifications, but these may be altered and extended by the user, and saved for future use.

1.2 Text editing

As all programs are ultimately displayed by ctedit's, text fragments edited outside of a ctedit must be parsed before pieces of abstract syntax tree may be redisplayed in a ctedit. After using any of the following text editing facilities, text is reparsed and, if correct, inserted at the current location.

- *Emacs* : Ctedit's may communicate with external editor processes such as emacs. Through the graphic interface, you may select a program fragment, send it to emacs, and from emacs, have the modified text parsed and sent back to the source ctedit.

- *In-line editing* : Rather than send text back and forth between ctedit and emacs, it is possible to edit fragments of text in place in a ctedit. You must enter a special in-line mode and type text directly in the ctedit (standard editing commands available) at a selected location. Leaving the in-line mode has the modified text parsed and, if correct, inserted at the current location.

- *X selection* : One may use the X selection to transfer text from any X window to a ctedit ; the text is parsed and, if correct, inserted at the current location.

2 Clipboards

Structured editing makes use of clipboards. A clipboard is a ctedit whose variable contains the result of the most recent edition of a source tree. Like any other ctview, the clipboard has a current selection. It is always the clipboard's currently selected subtrees that are the object of the editing command.

2.1 The global clipboard

The *global clipboard* receives the last editing results from a source variable. This clipboard is shared by all ctview's, which allows you to pass syntactically appropriate subtrees from one editor to another. To display the global clipboard, click on the *Clipboard* button of the toplevel menu.

2.2 Local clipboards

Each ctview may also create a *local clipboard.* Unlike the global clipboard, a local clipboard receives and sends *copies* of subtrees and is only accessible by one ctedit at a time.

When opened from the *Clip* button of a ctview's **Edit** menu, the local clipboard contains a copy of the currently selected source subtree(s). Changing the source current selection does not affect the clipboard current selection. To update the clipboard's contents to the current source selection, reclick on *Clip*. To achieve this through mouse interaction, include the package {**ctedit**}**:clip** in a mouse event list specification.

```
Centaur.Editor.MouseEventList: ((middle {ctedit}:clip))
```

With the previous resource, clicking with the middle mouse button is equivalent to using the *Clip* button of the ctview's **Edit** menu.

The source editor listens to changes in the local clipboard. When you change the clipboard's current selection to a new subtree, a copy of this subtree replaces the source's current selection. This default behvior may be modified to allow for insertion before or after the current selection by specifying the mouse packages `{ctedit}:current:insert+` or `{ctedit}:current:-insert-`. You may search for the next occurrence of an item in the local clipboard by specifying the mouse package `{ctedit}:current:find`.

N.B. A local clipboard may be unlinked from one editor and linked to another, allowing the transfer of subtrees as with the global clipboard. This is done with the *Link* pulldown.

2.3 Auto-clip

Auto-clip is a clever editing mechanism whereby the ctview uses itself as a clipboard. This is done by means of a special selection that keeps track of trees. A standard variable (the subject of a ctedit) has a `{current}` selection that designates subtrees of the variable's root. Since a variable may manage any number of selections (see the CENTAUR path manual), we add one named `{current}:clip` to designate a different subtree, thus simulating a clipboard.
There are two ways to exchange subtrees between the current and clip selections.

- Click to set the auto-clip selection. The current selection subtree is also replaced by the auto-clip selection subtree (syntax permitting). For this mode, bind the package **{ctedit}:clip:auto-clip**.
- Click to set the current selection. The current selection subtree is also replaced by the auto-clip selection subtree (syntax permitting). Note that you must have bound the auto-clip selection first. For this mode, bind the package **{ctedit}:current:auto-clip**.

The following resource specification sets middle-shift to the first mode and left-shift to the second mode. Other bindings are not shown:

```
Centaur.Editor.MouseEventList : ((middle-shift . {ctedit}:clip:auto-clip)\
(left-shift . {ctedit}:current:auto-clip))
```

3 Template editing : The menu mode

The *menu mode* enhances a language environment by allowing users to take full advantage of CENTAUR's structured editors, reducing programming errors while increasing programming speed and comprehension. A menu is characterized by its:

- *type* : Each menu represents a task. A menu entry is an option for completing the task. For example, if a menu's task is to substitute text in an editor, its entries are valid text substitutions. Selecting a menu entry performs the task.
- *context* : Valid menu entries are calculated according to a context and computation method. Contexts may be syntactic, semantic, or whatever you can think of. Typically, syntactic context is the phylum of the currently selected subtree determined by the corresponding position in the parent operator's signature. Menu entries are updated automatically at every context change, ensuring continually valid menus.
- *presentation* : Once menu entries have been computed, you may control their presentation through pretty printing specifications.

A programming language environment usually offers both structured and non-structured editing functionalities. Structured editing ensures syntactic correctness whereas unstructured editing allows more freedom. Just as unstructured editors such as GNUEMACS allow you to define special modes for various file types (lisp mode, TeX mode, etc.), the menu mode allows the customization of a language environment through menus that reflect common tasks. This accelerates editing and reveals essential aspects of the language.

The primary application of the menu mode in CENTAUR is **guided editing mode**, which offers syntactically sensitive menus to aid programming. Several menus are available for common tasks such as template editing and program transformations. You may toggle between the beginner and expert modes, whose menus present more or less detailed entries, respectively. You may also define editing shortcuts, customize mouse behavior for specific menus, and save menus for future CENTAUR sessions.

Notes

A menu is associated with a ctedit. To associated it with another ctedit, click on the *Attach* button in the *Link* pulldown of the box and then click on the desired editor. You may disconnect it from a ctedit with the $\boxed{\textit{Detach}}$ button. This may be useful if one wishes to switch back and forth between several menus for different ctedits.

3.1 Guided editing mode

CENTAUR's structured editors guarantee syntactic correctness. Structured editing applies to trees of any formalism, but as such, does not reveal the important features of a given formalism. As a result, users not familiar with a language's syntax may not find these simple operations sufficiently informative. The guided editing mode allows one to ensure syntactic correctness and rapid development through meaningful transformations.

By default, guided editing mode functionalities are available through the $\boxed{\textit{Editing-Tools}}$ pulldown of a ctview (see section 3.2 for details on menu mode resources). This pulldown contains all of the buttons described below.

3.1.1 Transformations

Rather than incessantly having to retype endlessly common language fragments, especially formalism key-words known by CENTAUR in advance, we can exploit *transformations* (ready-to-use program patterns). Selecting text in a CENTAUR editor establishes a syntactic context (a phylum) and displays all transformations corresponding to this context in an associated menu. By clicking with the mouse, the user may either replace the current selection by applying one of the available transformations, or insert a transformation before or after the selection. When the user selects atomic or annotation values that are trees from another formalism, the menu adjusts automatically to the new formalism.

Notes

Transformation editing is never mandatory; you may freely intermix it with other editing mechanisms.

Beginner and expert guided editing

There are two guided editing modes, beginner and expert, which differ in the detail of information displayed for each transformation. In beginner mode, a transformation is pretty printed as the signature of the associated operator. In expert mode, only the transformation name is displayed. When a transformation rule is an operator template, the transformation name is the operator's name.

To edit in beginner mode, click on the *Beginner* button. The transformation menu appears and displays the phylum context for the current selection and all patterns available for this phylum. If there is no selection, all possible patterns of the language, as defined by the `EVERY` phylum, appear in the menu. This menu applies to the editor that created it.

Clicking on a transformation name or body replaces the source selection with the template or inserts the template before or after the current selection, according to the current mouse bindings (see section 3.2). If the menu offers only one template, CENTAUR automatically replaces the text with the template. Clicking on the phylum itself replaces the current selection with a metavariable whose name is the phylum. Is is possible to cancel a choice by releasing a pressed button outside the menu window.

For example, in a TYPOL program, if we select a subtree whose operator belongs to the **RULE** phylum, we would see the beginner transformation menu shown in figure 1.

You may not, however, edit atomic values with the transformation menu. You must either enter them using cuts and pastes, using a text editor (section 5), using in-line editing (section 6), or using the atoms menu, described in section 3.1.1.

N.B. It is possible to construct a program in an empty CENTAUR editor using the guided editing mode. Click on *Beginner* (or *Expert*), and enter the language name in the dialog box that appears. The transformation menu may be killed and reopened at any time. It may be connected to a different ctedit with the *Link* pulldown.

Adding transformations to the transformation menu

Often, one wants to tailor a menu to a particular programming framework by adding frequently encountered transformations. The transformation menu is itself a CENTAUR structured editor and may be modified in several ways:

- You may add a pattern to the transformation menu by selecting a subtree in the parent editor and clicking on the *Add Transformation* button. The selected subtree appears as is in the transformation menu for each phylum containing the selected operator —the left hand side of this transformation is invisible since the transformation applies for

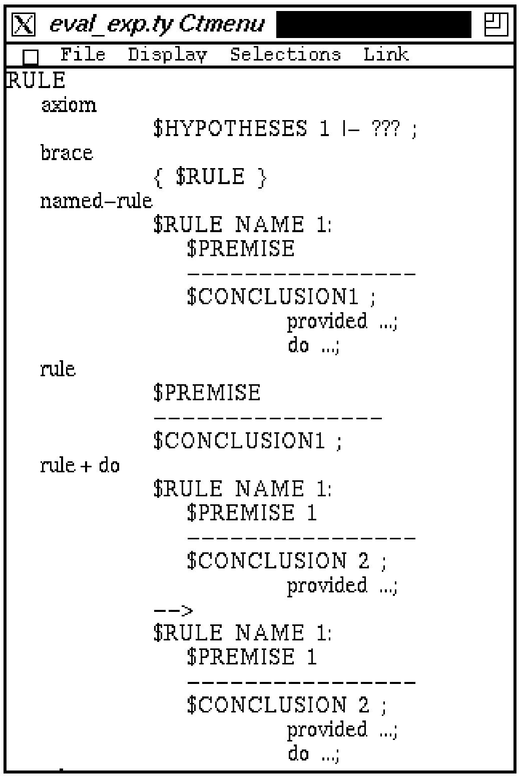

Figure 1: Excerpt from the beginner mode transformation menu for a TYPOL rule.

any phylum containing the operator. Instead of clicking on the button, you can add a pattern just by selecting a subtree. To do this, include the package **{ctedit}:add-transformation** in a mouse event list specification.

- You may specify transformations using the TransForm formalism and add them to the menu. Please consult section 4 for details.

N.B. You should not edit a menu by saving it in an external file then rereading and editing it. See section 3.3.

Editing atomic values

To edit atomic values in guided editing mode, open the atoms menu by clicking on the $\boxed{\textit{Show Atoms}}$ button. This menu is only active when either the current selection designates a tree with an atomic operator or you have selected an atomic operator from the transformation menu. You may click on a value from this menu to replace the currently selected value. The atoms menu only displays those values having the type specified by the atomic operator definition *and* that currently appear in the program. The atoms menu may be killed and reopened at any time.

To add new values (e.g., a new variable name or a new constant value), you may:

- use in-line editing.
- use cut and paste (structured or X selection).
- click on the `<Edit>` entry of the atoms menu and edit the text in GnuEmacs.[1] Once you have reinserted the value in the atoms menu, you may select it to insert it in the source ctedit.

For singleton atoms, the current selection is automatically replaced by the value of the operator.

N.B. You may not textually edit the values of *gate* atoms, i.e., atoms with tree values. However, if you click on a gate atom in the transformation menu, Centaur prompts you for a formalism name, and then updates the transformation menu for the new formalism. This is in fact the only technique for editing gate atom values.

[1] This button calls a generic parser, not the language's parser, so programs may be written before the language parser exists.

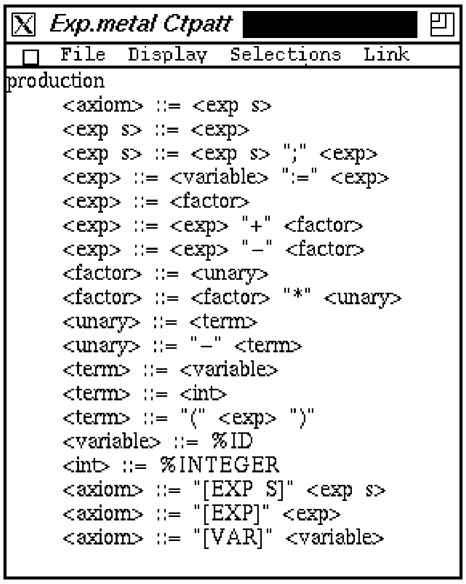

Figure 2: An occurrences menu showing subtrees with the `production` operator of the METAL formalism.

3.1.2 The occurrences menu

When you select an atom in the source editor, the atoms menu displays all atoms that occur in the source program. The occurrences menu functions similarly for non-atomic operators. It displays all subtrees in the source program with the same operator as the currently selected subtree(s). The displayed trees may contain metavariables.

You may open the occurrences menu by clicking on the *Show Occurrences* button. Figure 2 shows the occurrences of the `production` operator in a METAL program.

3.2 Customization

The following section describes guided editing mode's mouse and menu customizations.

3.2.1 General customization

The following resources regulate general menu mode behavior. They should be included in the user's `.centaur.rdb` file. For these resources, the query prefix is:

```
< <centaur-name>.menumode | Centaur.Menumode >
```

The following resources, preceded by the query prefix, may be specified:

Allmenus: When **t**, every new ctview sports an $\boxed{\textit{Editing-Tools}}$ menu. When the resource is **none**, this menu is not created. The default value is **t**.

Leftbigbox: When **t**, the calculator box appears automatically in the left column of new ctviews. When the resource is **none**, the box is not automatically created (the $\boxed{\textit{Add Box}}$ button still allows you to create it). The default value is **none**.

Popmenu: When **t**, all menus pop-up automatically when updated. They do not pop when the resource is **none**. The default value is **t**.

Gotometa: When true, the current selection is automatically moved to the next metavariable of the edited text, after any modification. The current selection is not changed if the resource is **none**. The default value is **t**.

For example:

```
Centaur.Menumode.Leftbigbox : t
mycore.menumode.Popmenu : none
```

3.2.2 Mouse customization

We specify mouse semantics through ctedit resources. Please consult the CENTAUR resource manual and the ctedit manual for details.

The guided editing mode menus offer several functionalities that may be triggered by the mouse:

- You may change the current selection in the source CENTAUR editor with a menu item (**change**).
- You may insert a menu item before (**insert−**) or after (**insert+**) the current selection, when permitted.
- You may search for the next (preorder) occurrence of a selected operator in parent editor (**find**).

These functions may be bound for three types of menus:

- *Ctmenu* : The transformation menu. Use the prefix `{ctedit}:menu`.
- *Ctatom* : The atoms menu. Use the prefix `{ctedit}:atom`.
- *Ctocc* : The occurrence menu. Use the prefix `{ctedit}:occ`.

For example, to specify that the left button of the *Ctmenu* triggers the function **change**, the middle button **insert+**, and the right button **find**, we write:

```
Centaur.editor-Ctmenu.MouseEventList : (\
      (left   .  {ctedit}:menu:change)\
      (middle .  {ctedit}:menu:insert+)\
      (right  .  {ctedit}:menu:find))
```

By default, each button performs a **change** for each menu type.

3.2.3 Menu resources

The menu mode resource manager object (rmo) is parented by the *Formalism* rmo. Thus, the query prefix for menu resource specifications is:

```
< <centaur-name>.formalism-<name>.menumode |
   Centaur.Formalism.Menumode >
```

The *Menumode* rmo properties are:

- *Beginner* : The name of the transformation menu to use in beginner mode.

- *Expert* : The name of the transformation menu to use in expert mode.
- *Atom* : The name of the atoms menu.
- *Occurrence* : The name of the occurrence menu.
- *Leftbigbox* : When the *Leftbigbox* has been specified globally to appear in all ctviews, this property allows you to remove it from ctviews for a given formalism. To do so, the property value should be **none**.

If no menus are specified by resources, they are calculated automatically from the abstract syntax information. To specify the beginner and expert mode transformation menus for the formalism `Exp`, for example, we write:

```
Centaur.Exp.Menumode.Beginner : mybmenu
Centaur.Exp.Menumode.Expert : myemenu
```

The *Menumode* rmo has four descendent classes: *Beginner*, *Expert*, *Atom*, and *Occurrence*. Instances of these classes are menu names. Below these classes or instances, we may specify values for the following properties:

- *Root* : Describes whether the menu is located in the system or user path. Possible values are **app**, **user**, and **abs**. **N.B.** Rather than specify a value for the *Root* property, we suggest specifying this property at the *Formalism* level with a wildcard. By doing so, the *Root* property of all resources concerning a formalism will be set by a single specification. Thus, a language hierarchy may be transferred from one root to another by modifying only one resource specification.
- *Location* : Describes the path to the menu's directory relative to the corresponding *Root* directory.
- *Load* : Indicates whether the Le-Lisp or polish representation of the menu should be loaded. Possible values are **lisp** and **polish** (the default value).
- *Pprinter* : For a source formalism `L`, indicates the `L` pretty printer to use to display the contents of an associated menu. The default value for the *Beginner* class is **begin**. The default value for the *Expert* class is **exp**. The default value for the *Occurrence* class is **occ**. The default

value for the *Atom* class is **atom**. **N.B.** In order for the guided editing mode to function properly, a specified pretty printer must be able to display metavariables.

For example, to specify information for the beginner mode transformation menu **mybmenu** we write:

```
Centaur.Exp.Menumode.Beginner : mybmenu
Centaur.Exp.Menumode.mybmenu.Location : contrib/Exp/menus
Centaur.Exp.Menumode.mybmenu.Load : polish
Centaur.Exp.Menumode.Beginner.Load : lisp
Centaur.Exp.Menumode.Atom.Pprinter : special_atom
```

which means that we load the polish form of the **mybmenu** menu but the lisp form of all other *Beginner* menus. We rely on a specification of the *Root* resource at the *Formalism* level. An atoms menu for `Exp` will be displayed with the `special_atom` pretty printer.

3.3 Saving menus

You may read and write menus in a flexible persistent format using the *Read.ll* and *Write.ll* buttons. The advantage of this representation is that even if source formalism changes you can still reread menus developed for it.

It is also possible to save a menu in a coded format (called the *polish* format) by clicking on the *Write* button of the menu's *File* pulldown. The polish format has the advantage of being very compact, but cannot evolve with a formalism specification.

4 Program transformation : The TransForm formalism

Many programming environments emphasize program transformations. When syntactic or semantic transformations can be fully automated, a special pretty printer or semantic tool may perform them. Some transformations require user interaction, such as choosing from several transformation options. The menu mode offers a generic selection mechanism that may be tailored to the transformation needs of a dedicated environment.

To specify menu entries devoted to program transformation, we rely on the TransForm formalism. A TransForm program is a set of optionally named transformation rules that describe how to restructure source abstract syntax trees that match a given pattern.

A rule consists of a pattern and a transformation. Both are fragments of the *abstract syntax* of the source formalism. A rule is considered a valid menu entry when its pattern matches the abstract syntax of the currently selected subtree. The pattern may be as specific as an instantiated operator signature, or as flexible as a metavariable constrained by one or more phyla.[2] If selected by the user, the pattern instantiates to the matched subtree and the transformation (which may contain metavariables of the pattern) replaces the subtree.

Transformations specified with TransForm always produce syntactically correct results since:

- the TransForm type checker first eliminates any erroneous patterns,
- the menu mode only offers patterns appropriate for the current syntactic context, i.e., menus do not offer transformations that would lead to syntax errors.

A TransForm program is compiled into Form menu entries that may be displayed by the menu mode.

4.1 Generating a TransForm specification

The TransForm environment enables you to generate a skeleton TransForm program from an abstract syntax specification. For each operator of the source abstract syntax, the program contains a rule whose:

[2] A phylum constraint means "any operator belonging to the phylum."

- name is the operator name
- pattern consists of a single unconstrained metavariable named ***any**.
- pattern is constrained by all phyla containing the operator
- transformation reflects the operator's signature (a metavariable for each descendent of a fixed arity operator, a list metavariable for list arity operators, and a metavariable for an atomic value).

Each rule may be specialized interactively by changing the pattern or transformation as desired. For example, you may wish to add several rules for a given operator with different parts of the pattern more or less instantiated. Or, since the default pattern is a metavariable whose constraint includes the `EVERY` phylum (which contains operators of the formalism and is usually too permissive a constraint), you may want to remove it.

Notice the difference between the pattern's constraint and possible constraints on metavariables in the pattern. The pattern's constraint is a list of phylum. Any metavariables in the pattern may be further constrained by a single phylum, making the pattern that much more specific.

The `transform-gen` function generates a complete TransForm program according to an abstract syntax specification and displays the resulting program in a new ctview.

4.2 Pattern matching

Rule patterns and transformations may range from simple to quite complex.

The simplest pattern is a metavariable. The pattern may be constrained by any number of phyla. The metavariable, which may also be constrained, is instantiated when the rule is selected, may also appear in the transformation.

The root operator of the transformation must also observe the pattern constraints.

Example. Consider the following excerpt from a Metal abstract syntax definition:

```
instructions -> INSTRUCTION * ... ;
if           -> COND INSTRUCTION INSTRUCTION ;
while        -> COND INSTRUCTION ;
assign       -> VARIABLE EXPRESSION ;
INSTRUCTION ::= instructions if while assign ;
```

The following transformation surrounds a list of instructions by a *while* statement:

```
'surround_while' : *inst_pattern : {INSTRUCTION}
             -->  while(*COND, *inst_pattern) ;
```

The `surround_while` rule matches any operator belonging to the phylum `INSTRUCTION`. In other words, the pattern constraint is the `INSTRUCTION` phylum. The metavariable has no constraint.
The transformation unravels as follows. Suppose that in the source editor, we have selected several elements of an `instructions` list. The transformation menu offers the above rule (and possibly others) since the `instructions` operator belongs to the `INSTRUCTION` phylum. If we select this transformation, the variable `*inst_pattern` instantiates to the selected list of trees. These trees are replaced by a single tree whose associated operator is `while`. The original list becomes the second descendent of the `while` tree. The first descendent of the new tree remains a metavariable tree.

Figure 3 shows a TRANSFORM program for the EXP formalism that contains generated rules and additional rules to commute, to factorize, and to distribute subterms of arithmetic expressions.[3]

4.3 The TRANSFORM environment

Once you have created a TRANSFORM program, you may edit it in a CENTAUR ctedit. The TRANSFORM programming environment includes several functionalities made available through the buttons of the TRANSFORM popup menu. By default, this popup appears when you click on the right mouse button.

- *Find Rule* : When you click on this button, a dialog box prompts you for the name of the rule to find. If such a rule exists, it is selected, otherwise a warning is printed in the main CENTAUR window.
- *Type Check* : This button verifies that no specified TRANSFORM rule will produce a syntax error when applied to a source tree. Errors (described in section 8.2) are displayed in a window entitled **environment**. To examine the source of a type checking error, click on the error message. No window appears if the program is error free.

[3] EXP is a language of mathematical expressions described in *The* CENTAUR *Tutorial.*

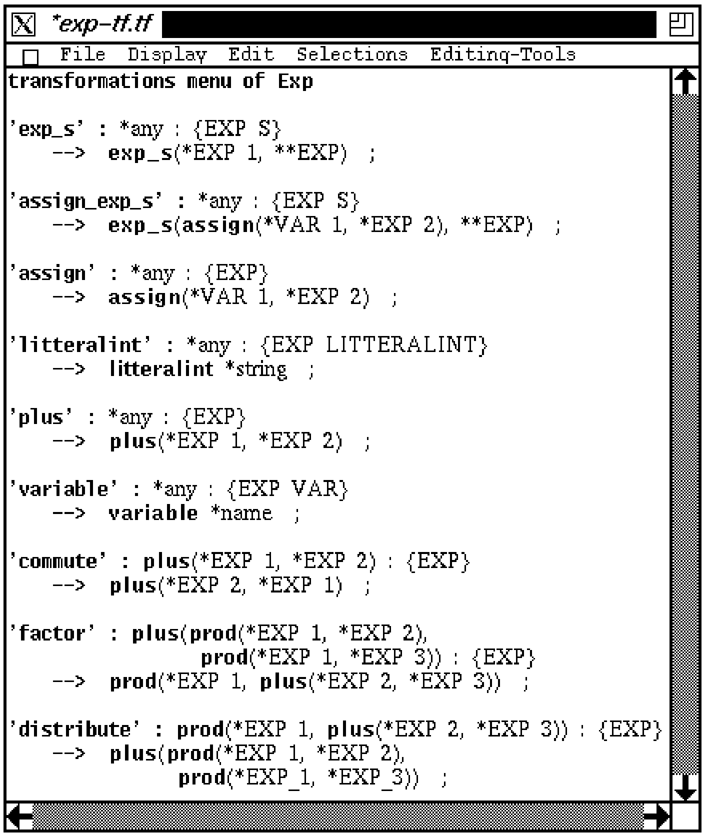

Figure 3: A TransForm program for the Exp formalism.

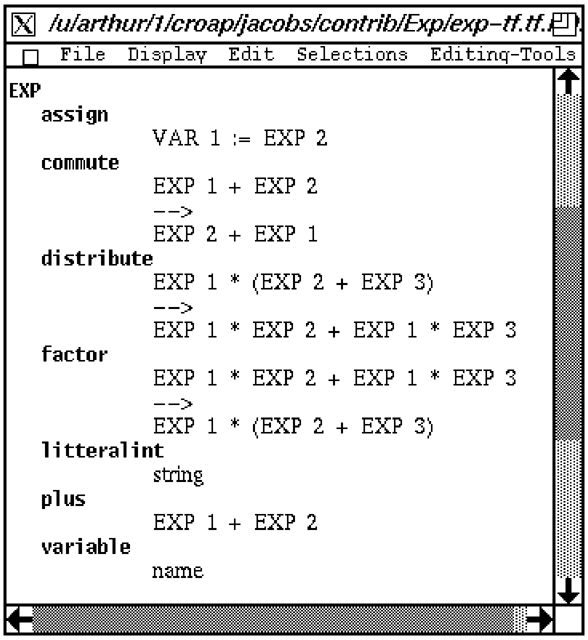

Figure 4: A menu generated from a TRANSFORM specification.

- *Compile* : Once type checking has succeeded, this button compiles a TransForm program into a Form menu. The Form program is displayed in a new ctview. This menu may be saved and reloaded automatically when specified by menu mode resources (see section 3.2).

In figure 4, we show an excerpt of a menu generated by the TransForm program of figure 3. See appendix 8 for more details.

5 Emacs

To enable unstructured editing, CENTAUR communicates with an external GNUEMACS process, possibly on another machine.

5.1 Installation

The user must first load the following file into a running GNUEMACS with the call:

```
(load-file "<centaur-root-dir>/etc/centaur.el")
```

This call may be included in the **.emacs** file of the user's home directory. The command:

```
(editor-start "<name>")
```

should be called from GNUEMACS, once CENTAUR is up and running, to create and initialize a special buffer with the given <name> which is either the name specified by the **-name** option of the command used to start CENTAUR, or by default, the command name itself. If at any time a communication problem interrupts the CENTAUR/GNUEMACS connection, you may repeat the **editor-start** command.

5.2 Connection with ctedit

To send text from a ctedit to GNUEMACS, select a program fragment in the ctedit and click on the *Edit* button of the *Edit* pulldown. The text appears in the buffer *<name>* and may be edited normally.[4] The buffer is "linked" to the ctedit that called *Edit*. Every click on *Edit* replaces the buffer's current contents.

To replace the current selection of the calling CENTAUR structured editor with the contents of the buffer, type **^x^s**. The new text is parsed by the formalism's parser (which is started up if necessary) If the text is syntactically correct, it sends it to the linked CENTAUR editor. When there is no current selection, nothing happens. The buffer remains intact, so you may select different locations in the tree and retype **^x^s**.

[4] If the selected fragment is not a full program, the name of the selected operator's phylum appears at the beginning of the text and should not be edited. The parser uses the phylum name as an entry point.

If the text sent from GnuEmacs is not syntactically correct, the buffer remains unchanged, the ctedit remains unchanged, and Centaur prints an error message.

Other available control sequences are:

- **^x^i** to insert the text before the current selection.
- **^x^a** to insert the text after the current selection.

As above, nothing happens when there is no current selection.

You may send a *complete program* to an initially empty Centaur structured editor by clicking on the *Edit* button as before (to establish the link) and then sending text from the buffer with the control sequence **^x^s**. A query box appears asking you for the formalism of the new program.

6 In-line edition

In-line edition involves typing text directly into a CENTAUR ctedit. A cursor indicates the current editing position.

The following table describes the default key bindings.

ˆe	enter in-line edition mode
Esc	exit in-line mode and parse what has been typed
ˆg	abort in-line mode
ˆf	next character
ˆb	previous character
ˆh	delete one character
ˆn	next line
ˆp	previous line
ˆa	beginning of line
ˆe	(once in in-line mode) end of line
ˆk	kill from the cursor to the end of the line

Leaving in-line mode causes the new text to be parsed and if correct, inserted in the tree. If the text is not syntactically correct, an error is triggered, and in-line mode remains active.

In order to use in-line mode, you must specify the proper ascii-handler in a mouse event list specification. The ascii package name is:

`{ctedit}:auto-menu:ascii`

For example, in your `.centaur.rdb` file you might include:

```
Centaur.Editor.MouseEventList: ((ascii . {ctedit}:auto-menu:ascii))
```

It is also desirable to be able to position the cursor with the mouse when in line mode. This may be accomplished with the package:

`{ctedit}:current:auto-scroll:line`

Thus, the previous mouse event list specification might become for ctedit's named `L` (which may be done in the environment setup for the formalism `L`):

```
Centaur.editor-L.MouseEventList: \
((ascii . {ctedit}:auto-menu:ascii)\
 (left . {ctedit}:current:auto-scroll:line))
```

6.1 Adding in-line edition to a user formalism

To be able to in-line edit a user formalism `L`, please follow these steps to create the **line** pretty printer.

- The `centaur-language` command automatically creates the **line** pretty printer directory for your formalism, so if you constructed your formalism hierarchy with this command, go to step two. If you did not create your language hierarchy with `centaur-language`, you may nonetheless call it on your existing hierarchy, entering no additional information. The script will create the line pretty printer directory and files without touching other files.

- Once you have created tables for your formalism `L`, you must compile the line pretty printer by going to the directory `L/pprinters/line` and typing `ctmake`.

- In the file `L/pprinters/pprinters.rdb`, there is a resource specification for the *Package* property of the display pretty printer. You may replace the default name `std` by the name of your principle pretty printer, add other pretty printers, etc.

- In the file `L/syntax/Exp.rdb`, a resource specifies that the display pretty printer will be invoked by default whenever you read an `L` file into a ctedit. This modular pretty printer allows you to perform in-line edition. Make sure the following resource is in this file and not commented:

```
?.Editor.Visual.Translator.formalism-L.Pprinter : display
```

7 The X selection

CENTAUR provides an interface between ctedits and the X selection mechanism. This allows one to transfer text between any X window and a ctedit. It also allows us to transfer between ctedits by way of strings, an easy way to pass identifiers between formalisms, you simply X copy from one ctedit and paste in another. Recall that inserted text will be parsed to ensure that it is syntactically correct. And unlike a normal X paste (e.g., in an xterm window), we provide several ways to insert text.

N.B. The size of an X copy/paste may not exceed 32 Kbytes.

7.1 Mouse behavior

The following functionalities may be bound to mouse buttons in mouse event list resource specifications:

- *copy current* : Copies the {**current**} selection (without changing it) to the X buffer. For this behavior, use the package `{ctedit}:xcopy` in a mouse event list specification.

- *set current and copy* : Sets the {**current**} selection and copies it to the X buffer. For this behavior, use the package `{ctedit}:current:xcopy` in a mouse event list specification.

- *change* : Parses the contents of the X buffer and attempts a change with the currently selected subtree. For this behavior, use the package `{ctedit}:current:xchange` in a mouse event list specification.

- *insert* : Parses the contents of the X buffer and attempts to insert the result before or after the currently selected subtree. To insert before, use the package `{ctedit}:current:xinsert-before`. To insert after, use the package `{ctedit}:current:xinsert-after`.

7.2 Resources

The following resources specifications address ctedit/X selection interaction.

In order to send a subtree from a ctedit to the X buffer, CENTAUR must convert it to a string. Resource specifications allow CENTAUR to pretty print a subtree as an ascii string. The X selection rmo is a subobject of the formalism rmo. Thus, resources for X selection properties begin with the following prefix:

```
< <centaur-name>.<formalism-name>.<formalism-name> |
  Centaur.Formalism.Xcopypaste >
```

The X selection rmo defines the following properties:

- `<pprinter | Pprinter>` : This symbol indicates the pretty printer to use when converting the current subtree to a string for the X buffer. The default value is **std**.
- `<context | Context>` : This integer indicates which context to use when converting the current subtree to a string for the X buffer. The default value is **0**.
- `<printLevel | PrintLevel>` : This integer indicates the global print level to use during pretty printing. The default value is **70**.
- `<width | Width>` : This integer indicates the page width to use during pretty printing, in characters. The default value is **255**.

For example, to specify that: 1) the left button sets and copies the current selection (for the `L` formalism, using the `special` pretty printer); 2) the middle button attempts to parse and paste the X buffer contents; and 3) the right button inserts the X buffer contents after the current selection, we specify:

```
Centaur.L.Xcopypaste.Pprinter : special
Centaur.Editor.mouseEventList(\
(left . {ctedit}:current:xcopy) \
(middle . {ctedit}:current:xchange) \
(right . {ctedit}:current:xinsert-after))
```

8 Appendix: TransForm reference manual

TransForm programs are usually generated from an abstract syntax specification. In this section we present the syntax and semantics of this language so that you may easily modify a generated TransForm program.

8.1 Syntax

8.1.1 Lexical elements

Identifiers

Syntax

$$
\begin{array}{rcl}
\mathit{identifier} & = & \mathit{lowercase\_id} \mid \mathit{uppercase\_id} \\
\mathit{lowercase\_id} & = & \mathit{lowercase}\ \{\mathit{alphanum}\}* \\
\mathit{uppercase\_id} & = & \mathit{uppercase}\ \{\mathit{alphanum}\}* \\
\mathit{alphanum} & = & \mathit{lowercase} \mid \mathit{uppercase} \mid \mathit{digit} \mid \text{“\_”} \\
\mathit{digit} & = & 0 \mid 1 \mid 2 \ldots \\
\mathit{lowercase} & = & a \mid b \mid c \mid \ldots \mid z \\
\mathit{uppercase} & = & A \mid B \mid C \mid \ldots \mid Z
\end{array}
$$

Strings

Syntax

$$\mathit{string} \quad = \quad \text{“ ' ”}\ (\{\mathit{no\_quote\_or\_newline}\}* \mid \text{“ ' ' ”})\ \text{“ ' ”}$$

Examples

`'st', 'string with spaces', ''''`

Phyla

Syntax

$$\mathit{phylum\_name} \quad = \quad \mathit{uppercase\_id}$$

Examples

`EXP, STATMENTS`

8.1.2 Metavariables

Syntax

metavariable	=	*simple_metavariable* | *typed_metavariable*
simple_metavariable	=	*anonymous_metavariable* | *single_metavariable* | *sublist_metavariable*
anonymous_metavariable	=	" *"
single_metavariable	=	" *" *variable_name*
sublist_metavariable	=	" **" *variable_name*
typed_metavariable	=	*simple_metavariable* " ~" *phylum_name*
variable_name	=	*identifier*

Description

The following rules govern pattern matching with respect to the current subtree's operator:

- An anonymous metavariable matches any operator. An anonymous metavariable is equivalent to the single metavariable named *any*.
- A single metavariable matches any operator and instantiates to the current subtree.
- A sublist metavariable matches the descendents of a list operator.
- A typed metavariable matches any operator belonging to the phylum named *phylum_name*.

Examples

```
*, *operator, **sublist, *typed~EXP
```

8.1.3 Patterns

Syntax

pattern	=	*simple_pattern* | *complex_pattern*
simple_pattern	=	*metavariable*
complex_pattern	=	*operator_name* | *node_pattern* | *atom_pattern*
node_pattern	=	*operator_name* " (" {*pattern*}* ")"
atom_pattern	=	*operator_name atom_arg*
operator_name	=	*lowercase_id*
atom_arg	=	*simple_metavariable* | *string*

Description

TransForm uses the same syntax as Ppml for describing simple patterns. The above patterns match the current source subtree as follows:

- *metavariable* : See section 8.1.2.
- *operator name* : Matches when the operator of the current subtree is the same as the pattern.
- *node pattern* : Matches when the operator of the current subtree is the same as the pattern and recursively for all instantiated parts of the pattern. Metavariables instantiate to corresponding subtrees.

8.1.4 Rules

Syntax

rule	=	*rule_name* " :" *typed_pattern* " `-->`" *transformation*
rule_name	=	*string*
typed_pattern	=	*pattern* " :" *type*
type	=	" {" {*phylum_name*}+ " }"
transformation	=	*pattern*

Description

A TransForm rule describes a transformation to be performed on a fragment of abstract syntax. When a rule appears in a menu, it is active (i.e., may be applied) when the abstract syntax of the current subtree matches the *typed_pattern* (see section 8.1.3) and the operator of the current subtree belongs to one the named phyla. When a rule is selected, the current subtree is replaced by the *transformation*, which may contain metavariables that appear in the *typed_pattern*.

Example: Transformation rule.
The following rule is the default rule generated for the `pprinter` operator of the Ppml formalism. It contains a simple pattern.[5]

[5] We leave the `EVERY` phylum in the pattern constraint to allow for the creation of a Ppml program in an empty ctedit.

```
'pprinter' : *any : {EVERY PPRINTER}
      -->  pprinter(*PPRINTER_NAME_1, *IDENT_2,
                    *INIT_3, *ZONE_S_4) ;
```

The rule may be modified to contain a complex pattern.

```
'pprinter-complex' : *any : {EVERY PPRINTER}
       -->  pprinter(*PPRINTER_NAME_1, *IDENT_2,
              init(constant(const_decl(*IDENT_1,
                                       *INT_CONST_2)),
              default(**DEFAULT_DECL)), zone_s(**ZONE))
```

Once compiled into FORM code, this previous transformation rule is displayed in a menu by the **std** pretty printer for the PPML formalism as follows:

```
PPRINTER
   pprinter-complex
          prettyprinter PPRINTER_NAME_1 of IDENT_2 is
          constant
              IDENT_1 = INT_CONST_2;
          default
              DEFAULT_DECL;
          ZONE
          end prettyprinter
```

Note that metavariables are shown by the `std` pretty printer without the preceding asterisk. The following example also refers to PPML.

```
'elem -> in class elem' : *elem : {ELEM}
-->  attribut_elem(attribut_s(attribut(ident 'class',
                                       *ID_INTEGER_2)),
          box(sep(ident 'h',
                 s_param_s(*dx)),
                 elem_s(*elem)))
```

The name of the rule describes the transformation (encapsulation of an element by a class description for the element). According to the metavariable name, we expect the operator of the current subtree to be `elem`, which belongs to the `ELEM` phylum. The transformation creates new pieces of PPML abstract syntax, reusing the subtree to which `*elem` instantiates. Note also that the tranformation introduces two metavariables (`*ID_INTEGER_2` and `*dx`) that do not appear in the pattern (and are therefore not instantiated). The generated FORM rule is displayed by this pretty printer as follows:

```
ELEM
  elem -> in class elem
     elem
     -->
     in class = ID_INTEGER_2 : [<h dx> elem]
```

Notes

- Rule order in a TRANSFORM program is not important.
- Rule names are significant. When compiling into a FORM menu, the rules appear in alphabetical order. Also, the name of each rule should be meaningful, because in expert guided editing mode, the menu displays only these names.

8.1.5 Programs

Syntax

program = "`transformations`" "`menu`" "`of`" *identifier* {*rule*}+

Description

The identifier of the program syntax is the source formalism name.

Example

```
transformations menu of Exp

'exp_s' : *any : {EXP_S}
   -->  exp_s(*EXP_1, **EXP) ;

...
```

8.2 Errors

This section documents TransForm type checker errors.

8.2.1 This TransForm definition contains metavariables

If the TransForm program has been edited using menus, it may still contain some metavariables. In this case, it is not possible to typecheck an incomplete program. No metavariables may remain in a program to be type checked.

8.2.2 This is not a valid TransForm definition

The typechecker accepts only TransForm programs. It is not possible to typecheck a rule or a pattern because the formalism of the patterns is needed.

8.2.3 Unknown language

```
transformations menu of L
...
```

Unknown language: L

The type checker fails on this error because Centaur is unable to find the tables for the formalism `L`. Please consult the Centaur resource manual for information about formalism location.

8.2.4 Unknown operator

```
foo(*x)
```

Unknown operator: foo

Operators referenced in a pattern must belong to the abstract syntax of the formalism whose name appears in the first line of the TransForm program.

8.2.5 Unknown phylum

```
    *x : {EVERY  PHYL}

     Unknown phylum: PHYL
```

Phyla referenced in a TransForm program must belong to the abstract syntax of the source formalism.

8.2.6 Operator does not belong to phylum

```
'pprinter' : *any : {RULE}
    --> pprinter(*PPRINTER_NAME, *IDENT, *INIT, *ZONE_S)

     Operator pprinter does not belong to phylum RULE
```

The top node operator of each pattern must belong to one of the phyla that appears in the *type* declaration in the lefthand side of the rule.

```
'pprinter' : *any : {PPRINTER}
    --> pprinter(*PPRINTER_NAME, *IDENT, *INIT,  rule_s(*RULE))

     Operator rule_s does not belong to phylum ZONE_S
```

The righthand side pattern is syntactically incorrect since the *rule_s* subtree doesn't belong to the *ZONE_S* phylum, as required by the signature of the `pprinter` operator.

8.2.7 This operator is not atomic

```
'rule' : *any : {EVERY RULE}
    -->  rule *patt

     This operator is not atomic: rule
```

Atomic patterns may only accompany atomic arity operators. The `rule` operator is not atomic.

8.2.8 Bad arity for this operator

```
'rule' : *any : {EVERY RULE}
    -->  rule (*patt)
```

Bad arity for this operator: rule

This transformation does not respect the arity of the *rule* operator.

8.2.9 Sublist metavariable is not allowed for a fixed arity operator

```
'rule' : *any : {EVERY RULE}
    -->  rule (*patt, **box)
```

Sublist metavariable is not allowed for a fixed arity operator: rule

Sublist metavariables may not be used with fixed arity operators such as `rule`.

8.2.10 A + list operator must contain one or more descendents

```
'rule_s' : *any : {EVERY ZONE}
    -->  rule_s ()
```

+ list operator must contain one or more descendents: rule_s

A + list operator such as *rule_s* cannot be empty.

8.2.11 This metavariable must belong to two phyla

```
'chapt-rule' : chapt(*chapter_name, *context , *init,
                     zone_s(rule_s(*rule))) : {ZONE}

    --> *rule
```

This metavariable rule must belong to two phyla ZONE RULE

This rule should transform a chapter containing a rule into this rule, but the *ZONE* phylum does not contain the *rule* operator: it contains the *rule_s* operator. The correct transformation is `rule_s(*rule)`.

8.3 Primitives and compilation

8.3.1 Interactive

(transform-gen formalism-name) $\rightarrow$ *tree*

Generates a TransForm program for the formalism named **formalism-name**. Displays the resulting program in a new ctview.

({transform}:form TransForm-tree) $\rightarrow$ *FORM-tree*

This function compiles the TransForm specification contained in the **TransForm-tree** (see the previous function) and returns a Form tree, corresponding to the specification.

8.3.2 Batch

In order to compile TransForm specifications in batch mode, you must create a directory that contains TransForm specifications and a Buildfile.[6] A TransForm specification is a textual specification stored in a file named `L-menuname.tf`. `L` is the name of the source formalism and `menuname` is the name of the menu that will contain the Form code resulting from the batch compilation (e.g., `beginner`, `expert`). The Buildfile should contain the following lines:

[6] Please consult the ctboot manual for more information on Buildfiles.

```
LANGUAGE=<language-name>
RDBFILES=<resource-file-name>/
POFILE=<language-name>-<menuname1>.po \
       <language-name>-<menunameN>.po
all: alltransform
#include ...../.CENTAUR_SRC/centaur/tables/transform/BuildTransForm
```

where `<resource-file-name>` is the path leading to the formalism resource file. This file should include resources that describe the *Location* of the formalism tables. Centaur looks for these tables first in the user hierarchy, then the application hierarchy. The five periods indicate a relative path to your home directory and `.CENTAUR_SRC` is a symbolic link to the current Centaur installation directory.

9 Syntax index

10 Function index

11 Error index

12 Concept index