A Centaur Tutorial

July 22, 1994

Centaur	version 2.0
Documentation revision	version 1.28 94/07/22 16:20:40 croap1 Stable

Contents

1 Introduction

In this tutorial, we walk through the development of an environment for a language of mathematical expressions called Exp. We highlight important notions along the way and explain all the steps necessary for creating an Exp environment.

1.1 Language specifications

First we specify the syntax of Exp with the Metal formalism. Compiling this specification produces an abstract syntax (formalism) definition, a parser, and tree building functions which the parser uses to construct Vtp abstract syntax trees from syntactically correct programs. All parsers generated from Metal specifications are external processes that communicate with Centaur.

Next we specify the pretty printer with Ppml, which we compile to produce an incremental tree formatter.

Finally, we specify an evaluator for Exp in Typol, which is compiled into a prolog program. The evaluator reads an Exp program and evaluates it in an environment consisting of variable-value pairs. The evaluator returns a modified environment.

1.2 Language environment

We develop an Exp environment in which:

- the evaluator may post warning messages which will be displayed in a special message display.
- users may call the Typol evaluator through a popup menu button.
- windows will display the initial and final environments. These windows will open by necessity and will close if the source window is closed.
- users may select a variable name in any display and see all occurrences in all displays with a special selection.

2 The Centaur environment

Aside from the specification languages, CENTAUR provides several generic tools that facilitate the development of an environment prototype.

We occasionally refer to the `<centaur-root-dir>` in this tutorial (and in the documentation). This directory contains the current CENTAUR installation. It contains the subdirectories `bin`, `lelisp`, `centaur`, `etc`, `contrib`, `gfxobj`, and `tools`.

2.1 Documentation

The CENTAUR documentation is divided into three sections: The Crust, which includes this tutorial; The Mantle, where you may find information about specification languages, the graphical interface, and editing; and The Core, which documents tools necessary for environment programmers. In case you are lost, the Core also contains a global function index, a glossary of CENTAUR terms, and a FAQ.

2.2 The VTP and paths

The Virtual Tree Processor, or VTP, is the heart of the CENTAUR system. The VTP is a set of tools that allows the specification and manipulation of abstract data types, called VTP classes. Several predefined data types such as trees and formalisms may be exchanged by different system components such as METAL, PPML, and TYPOL programs.

The VTP is a rather low level interface, and fortunately, most VTP primitives need not be manipulated directly; formalisms are generated by syntax specifications, PPML requires only operator patterns to function, etc. However, for certain tasks such as dynamically extending an abstract syntax with annotations or retrieving atomic values in a Le-Lisp program, we must program with the VTP.

Another important type of object that allows different CENTAUR components to communicate information is the *path*. A path describes locations in a tree. We use paths primarily to designate locations (e.g., for edition) or memorize information (e.g., error locations).

2.2.1 The language tables

It is possible to save a formalism in a persistent format known as the *tables*. Although a formalism may be created, saved, and restored by hand, we usually generate the formalism through an abstract syntax specification. Resources indicate where to store the tables when generated. These same resources allow CENTAUR to load the formalism the first time you try to construct an abstract syntax tree for the given formalism.

2.3 The CENTAUR variable

In order to allow many different tools (editors, pretty printers, semantics tools) to manipulate a common abstract syntax tree, we usually manage the tree with a CENTAUR *variable structure.* The variable has several roles:

- Allow tree edition.
- Keep track of paths that memorize information about the tree. Within a variable, a path is encapsulated within a *selection.* A selection's type determines how that path is updated according to tree modifications.
- Keep track of modification histories for incremental tools. Tools that maintain their own representation of the abstract syntax tree may update this representation incrementally with the modification information provided by the variable.
- Keep track of undo information.

2.4 Resources

CENTAUR features a *resource manager* so that values for most system parameters (e.g., formalism locations, pretty printer colors and fonts, mouse event lists, etc.) may be specified by resources. These resources are stored in resource files that are read by CENTAUR into its resource database by necessity. Resources may be modified dynamically within a CENTAUR session.

A resource specification is a pair that assigns a value to a pattern. Each pattern represents the path leading to an object in the system hierarchy. The last word in the pattern is a *property* of the object for which we may furnish a value, such as the background color used by a pretty printer or the location of a formalism. Patterns are modeled after those used by the X window system.

Although resources are omnipresent in CENTAUR and any serious user must get to know them, we will avoid them as much as possible in this tutorial. We will explain them briefly when we examine pretty printer colors and fonts.

Resources are stored in a number of files that are read into the resources database as required. One file, named `.centaur.rdb`, should be created in your home directory. It is always read when you start up CENTAUR.[1]

Another important file is the *formalism resource file*. This file specifies the location of the formalism definition, syntax and semantics information, and points to all other specification files for the formalism.

N.B.

- Any time you modify a resource specification in a file, you must click on the *Reset Resources* button in the **Load** pulldown of the main menu. This loads the resources, but does not inform tools of the new values. Typically, you will have to click on the *Redraw* button of a ctview's **Display** pulldown to take the resources into account.

- If ever resources don't do what you think they should, then verify the following:

  – You have not misspelled any words.

  – You may split a resources onto two lines by ending the first line with a backslash. Make sure that this backslash is not followed by blanks.

  – Make sure that the resource path you have specified is the right one, i.e., all the documented elements in the path are present and correct.

  – Make sure that your resource is the *most specific* one in the database (see the section on primitives in the resource manual of the Mantle).

2.5 Graphical interface

2.5.1 The main menu

The main menu consists of three pulldown menus:

- **View**: Contains buttons to open CENTAUR editor views or the global clipboard.

[1] Unless you specify otherwise on the `centaur` command line.

- **Load**: Contains buttons to load formalism tools such as the formalism's persistent representation, pretty printers, parsers, the environment module, etc. The *Reset Resources* button reloads resources from files.

- **End**: Contains a button to end the CENTAUR session.

2.5.2 The ctview and ctedit

The *ctview* is a viewing tool which allows us to manipulate programs through a graphical interface. Clicking on the *Editor* button in the **View** pulldown of the CENTAUR main menu opens a new ctview which contains an empty CENTAUR editor, called a *ctedit*. The ctedit is in charge of displaying information and communicating with the X server (thus, it handles mouse events, keyboard events, etc.) The ctedit contains two items: its *subject*, which is the thing being edited, and its *formatter*, which determines the subject's visual presentation. In this tutorial, the ctedit's subject is a CENTAUR variable structure.[2] The formatter will be the PPML formatter, which displays a syntax tree textually.

Each ctview menu bar has the following pulldown menus:

- *File*: This menu allows you to read programs from external files into the ctedit, and to write them to files.

- *Display*: This menu allows you to modify certain pretty printing parameters (e.g., print level, page width), change pretty printers, and refresh the ctedit contents.

- *Edit*: Allows you to perform generic edition: cut, copy, paste, or send selected subexpressions to a separate GNUEMACS editor for unstructured edition. The undo button of this menu undoes the last editing operation only.

- *Selections*: Once you have read a program into a ctview, you may select program fragments by clicking and dragging with the mouse. This menu allows you to fine tune the presentation of selected material. You may make shrink it (set its local pretty print level to zero), expand it (the opposite), or clear it.

[2] A variable may be the subject of any number of ctedits, but a ctedit may only have one subject at a time.

- *Editing-Tools*: This menu gives access to guided editing tools. We describe guided editing below.

When we read a program written in `L` into a ctview, CENTAUR loads the `L` environment which typically decorates the ctview with a graphic interface for `L` tools. If we have described a graphical environment for `L`, the ctview is transformed by the addition of the environment's buttons, menus, mouse behavior, etc. The enclosed ctedit becomes a structured editor that only allows edition according to `L`'s syntax definition.

2.5.3 Select

Once a program has been read into a ctedit, it is possible to select parts of the tree, for editing purposes, or other environmental needs as we will see later on. A *select* is the ctedit's way of displaying a CENTAUR variable's selection information. A select is only visible (and thus the selection it refers to) when the correct select resource indicates its background and foreground colors and priority.

The *current* selection is used by CENTAUR to designate a current subtree or subtrees. It is used for pointing and editing. In the environment section of this tutorial, we will add other selections to the CENTAUR variable when it contains an EXP tree.

2.5.4 Mouse and keyboard behavior

The *mouse event list* is a ctedit property that specifies mouse and keyboard behavior. The value of this property is a list of pairs. The car of each pair is the mouse button/keyboard combination. The cdr is the package of functions that will be called on mouse down, up, and drag events. All mouse behavior may be modified by modifying the appropriate resources.

2.6 Editing

CENTAUR offers a wide variety of structured and textual editing techniques. For any supplementary information about editing techniques, please consult the editing and transformation manual of the Mantle.

The mouse and keyboard behavior of the following descriptions may be modified by resources. The bindings required for the behavior indicated are given in section 3.1.1.

2.6.1 Reading a file

You may either read a program from an external file into a ctedit by clicking on the $\boxed{\textit{Read}}$ button of the **File** pulldown, or build the program from scratch using guided editing mode.

If you read an external file whose suffix doesn't provide information about which parser to call, CENTAUR will ask you for a language name. The dialog box contains a history of executed Le-Lisp commands, so you may recover previously entered information by typing `<tab>`.

If the file you read has any lexical or syntax errors, the parser indicates the line where the error occurs in the CENTAUR toplevel. You must correct all errors before a file may be read into the CENTAUR editor. If there are no errors, the text of the file appears in the editor window.

2.6.2 Structured editing

Structured editing is based on an abstract syntax specification. It ensures syntactic correctness since only complete syntactic elements can me 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). When you delete or change a subtree, the old subtree is stored in the *global clipboard*, which may be made visible by clicking on the $\boxed{\textit{Clipboard}}$ button of the main menu's **View** pulldown. Subtrees that you paste are taken from this clipboard. The editing and transformation manual also describes how to use local clipboards to accelerate edition.

- *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.

The primary application of menu mode is the *guided editing mode.* This tool eliminates syntax errors and takes advantage of information inherent in the syntax to accelerate program construction. In this mode, clicking on an applicable template replaces the currently selected fragment with the menu entry.

To experiment with guided editing, open a new editor by clicking on the *Editor* button of the **View** pulldown in the main menu, then click on *Beginner* in the **Editing-Tools** menu. A new window opens up, followed by a dialog box asking you for the formalism name of the program to be developed from scratch. Enter `METAL`, in all capital letters. Several things happen:

- The ctedit contains a METAL program fragment. This program is not whole; it contains *metavariable trees*, which are place holders. Guided editing is the process of gradually instantiating these metavariable trees with more and more detailed program fragments.

- The second window is the beginner mode menu. One may select program fragments from this menu (and others) during guiding editing.

The beginner menu's contents change as the current selection in the source editor changes. The currently selected subtree establishes a syntactic context, namely a phylum. This phylum appears in capital letters in the beginner menu, followed by patterns for operators that belong to this phylum. Each operator name is followed by its signature, where descendents are metavariable trees whose names are the phyla of valid descendents. Clicking on an operator name in this menu replaces the currently selected subtree by the operator's pattern. This pattern is in fact a subtree that is pretty printed in the source ctedit using the current pretty printer. You may go on replacing non-terminals this way to build your program. The expert menu is just like the beginner menu except that it displays less pattern information.

To replace atomic values, click on the *Show Atoms* button of the **Editing-Tools** menu. This menu contains all atomic values and tokens that currently appear in your program for the selected context. Clicking on the `<Edit>` entry of this menu sends the tree or metavariable tree text to GNUEMACS for edition. Or, you may click on an existing atoms menu entry to replace the current selection with that atom.

2.6.3 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 GnuEmacs. Through the graphic interface, you may select a program fragment, send it to GnuEmacs, and from GnuEmacs, have the modified text parsed and sent back to the source ctedit.

  To connect GnuEmacs with Centaur, first you must include the following line in your `.emacs` file to enable the Centaur/GnuEmacs connection:

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

  Then, open a special Centaur buffer in GnuEmacs by typing Escape-x, then `editor-start`. You are asked to enter the name of the Centaur session. By default, this name is `centaur`, so you may just type a carriage return. Then, to send the currently selected subtree from a ctedit to GnuEmacs, click on the *Edit* button of the **Edit** pulldown. If you have selected less than the entire program, the phylum name appears between brackets at the head of the text. The phylum name must not be edited. You may edit the text and send it back *to replace the current select* by typing `^x^s`. If a parsing error occurs, the text remains unchanged in the ctview. You may also insert the new text before or after the current selection with the keystrokes `^x^i` and `^x^a` respectively.

- *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. To enter in-line editing mode, type `^e`. You may then edit the current selection with GnuEmacs-like keystrokes to go to the beginning of the selected text, cut until the end of the line, etc. To leave and reparse the modified text, press the **Escape** button.

- *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.

2.6.4 Editing tips

Here are a few editing tips:

- By default, the beginners menu only contains fragments of the abstract syntax as specified by the formalism. However, you may add frequently used patterns (more or less instantiated) to this menu by selecting an expression in the source ctedit then clicking on the *Add Transformation* button of the **Editing-Tools** menu.
- Use metavariables as points of reference. For example, rather than programming a statement to print "Not implemented," just leave a metavariable at that position. You can go back to it later. And, you may use the calculator box (popped up by the *Add Box* button of the **Editing-Tools** menu) to navigate directly to metavariables within a program.
- Use guided editing mode (since much syntax information is supplied for you). Use in-line editing on small subtrees, typically those that are printed on less than one line.
- Use clipboards to store frequently used subtrees. Please see the editing and transformation manual for information about local clipboards.

2.7 Error handling

Language environments may use a generic mechanism for displaying messages. The errors generated by running a tool (e.g., a type checker, compiler, etc.) appear in the source ctedit (as selected information) and in a special message display (as text). Clicking in the message display on a message argument with the left mouse button highlights the message's corresponding position in the source (if a position has been specified). You may then flip through the error messages in the order that they appear in the source by clicking on the *Show next* and *Show prev* buttons in the left column of the message display. Only "active" messages are displayed in the source. Active messages are displayed in color in the message display. You make make all or some messages "inactive," removing them from the source and making them gray in the message display. This allows you to examine only a few errors at a time and reduce clutter in the source editor. Or, if several tools display messages at the same time, you may want to render one tool's messages inactive while you examine the other's messages. The *Hide All* and *Show all* buttons of the message display make all messages inactive or active. Clicking on a message with the left mouse button while holding the shift button down in the message display toggles the message back and forth between active and inactive.

N.B. Syntax errors encountered while parsing programs appear in the CENTAUR toplevel and in a general Le-Lisp error window (which if closed, is reopened automatically). Syntax errors prevent the construction of an abstract syntax tree, so no tree appears in the editor.

2.8 Parsers

CENTAUR starts external parsers for any known language on demand, according to the system resources. These parsers continue to run even when you exit CENTAUR so that you may use them for another session.

N.B. If you create a new parser in batch mode (from unix), CENTAUR will not be aware of the change. You must kill the old parser and start the new one by clicking on the *Load Syntax* button of the **Load** pulldown of the main menu.

3 Preparation

Before attacking the design of the Exp environment, we must make a few preparations.

3.1 The language hierarchy

The language hierarchy is the tree of directories and files that make up the language environment. Although you may create a language hierarchy anywhere you like, we suggest first creating a directory:

```
$HOME/contrib
```

that will contain all language hierarchies. We call this directory the contrib directory. To create the Exp hierarchy, go to the contrib directory and type after the unix prompt (here `%cahors`):

```
cahors% centaur-language Exp contrib
```

This command creates almost all the files necessary in a language hierarchy. You will be asked for information concerning:

- *pretty printers*: Enter the names `basic env`.
- *semantics tools*: Enter the name `interpreter`.
- *message modules*: Enter the name `interpreter`.

N.B. This command indicates a few lines of resources that must be included in the `.centaur.rdb` file of your home directory:

```
Centaur.Formalisms: (... Exp ...)
...
Centaur.Exp*Root: user
Centaur.Exp.Database.Location: contrib/Exp/syntax/Exp.rdb
```

where ... indicates the possible presence of other information.

3.1.1 Mouse event list

In this tutorial we wish to adopt the following behavior for ctedits that contain EXP trees:

- Clicking with the left button sets the current selection for editing purposes. Also, dragging the mouse outside the ctedit window with the left button depressed scrolls the ctedit.
- Clicking with the left button while holding the shift button copies the designated subtree (without setting the current selection) to the X selection buffer. Clicking with the middle button while holding the shift button replaces the designed subtree with the contents of the X selection buffer.
- Clicking with the middle button on a variable name highlights (with a special selection) all occurrences of the variable in all windows.
- Clicking with the right button pops up a menu that contains buttons to start interpretation and to clear the initial and final environment views.
- Typing ctrl-e enters *in-line editing* mode.

To achieve this behavior, we must specify an appropriate *mouse event list* for any ctedit containing an EXP program. The following lines should be included in the file:

`contrib/Exp/syntax/Exp.rdb`

```
Centaur.editor-Exp.mouseEventList:(\
        (left . {ctedit}:current:auto-scroll) \
        (left-shift .{ctedit}:current:xcopy) \
        (middle-shift . {ctedit}:current:xchange) \
        (middle . {Exp}:select-variable) \
        (right . {ctedit}:popup) \
        (ascii . {ctedit}:auto-menu:ascii))
```

All of the above callback packages except {`Exp`}`:select-variable` are predefined by CENTAUR. We explain the new package in section 7.

Example. To replace the current selection with a menu entry, click on the menu element with the left button. To set the current selection and simultaneously copy it to the X selection buffer, press the shift key and click with the left button. To replace the current selection with the contents of the X selection, use shift/middle button.
N.B. The preceding resources *only* apply to those editors named `Exp`. In section 7 we show how to assign a name to any editor containing an EXP tree.

Finally, make sure you have correctly positioned the unix environment `DISPLAY` variable so that CENTAUR appears on the correct machine.

3.2 Calling Centaur

We may now call `centaur`. To call a unix command from within CENTAUR, type an exclamation mark followed by the command[3].

3.2.1 The edit-compile-reload cycle

In this tutorial, as in a real CENTAUR session, you will find yourself following a cycle of program editing, compilation, and reloading. When you change a formalism definition, pretty printer definition, environment, etc., you should reload the appropriate tool from a button in the main menu's **Load** pull-down. Thus, after compiling a METAL abstract syntax specification, click on *Load Formalism*. A menu will display all known formalism names (as declared by resources). You may click on *Cancel* should you decide against reloading.

And remember, if modify a file containing resource specifications, click on *Reset Resources* then *Redraw* in the ctview's **Display** pulldown.

[3]When changing directories with `cd`, no blanks may appear between the lisp prompt and the exclamation mark.

4 The Metal definition

4.1 Introduction

We begin our language environment by specifying the syntax of the Exp language with Metal. The Metal specification for the Exp language will be stored in the file:

`contrib/Exp/syntax/Exp.metal`

This directory has been specified by the *Location* property in the `Exp.rdb` file of the same directory.

For further information about Metal, please consult the Metal manual of the Mantle. The parsers manual of the Core describes parsers in more detail.

4.1.1 Abstract syntax

We define a language's grammar as an *abstract syntax* specification. A hierarchy of grammar rules delimits a set of legal abstract syntax trees that may be created. An abstract syntax definition consists of nodes, called **operators**, and node types, called **phyla**. The operators are the terminals and non-terminals of a language. An operator is defined by the type and number of its descendents. An operator may belong to more than one phylum, meaning that it may appear in different contexts. In our example of a language of mathematical expressions, we may define a phylum `EXP` which contains all the operators of our language:

```
EXP ::= plus minus prod uminus assign variable integer ;
```

We include the assignment operator and integers in our language of expressions. Note that `plus` is a binary operator whose descendents also belong to the `EXP` phylum. An operator's *signature* is defined by the number and phyla of its descendents. An abstract syntax thus recursively defines operators, their descendent phyla, the operators contained by these phyla, and so on.

Operators come in three flavors:

- *fixed arity* : A fixed number of descendents of any type.
- *list arity* : Any number of descendents of the same type. A `+` list must contain at least one element, whereas a `*` list may be empty.

- *atomic* : No descendents, but we may associate a typed value.

In METAL, phyla names must begin with an upper case letter followed by any number of letters, underscores, or numbers. Operator names must begin with a lower case letter followed by any number of letters, underscores, or numbers.

In an organized abstract syntax specification, each phylum gathers related operators together, forming a kind of language subset. The phylum `EXP` for example, contains all of the operators necessary to represent a single mathematical expression. Due to the recursive relationship between phyla and operators, even a moderately sized language quickly becomes an intricate hierarchy of language subsets. When attacking a relatively important language, it is indispensable to organize language elements modularly, i.e., through phyla.

Given an abstract syntax definition, the METAL compiler generates a persistent representation called the *tables*. Whenever we try to construct an abstract syntax tree in a given formalism, either by hand or by parsing, the tables are read into memory (unless already present) and serve as the basis of tree construction.

We implement comments in this version of EXP. Comments pose an interesting problem in a language specification. We would like to insert comments virtually anywhere in a program. In order to preserve this position information, we may store comments in one of two ways. First, each operator signature could contain a comment descendent. But this would be very cumbersome and detract us from the real signature of the operator. Instead, we store comments as *annotations* on the source tree. A comment is attached to an abstract syntax tree at a node constructed from nearby text. Whereas we specify an operator's signature statically, we may also dynamically extend abstract syntax for requirements such as comments. The METAL compiler automatically includes comment operators and phyla in the formalism tables, so to include them in EXP, we need only specify their concrete syntax (see section 4.5.2.)

4.1.2 Concrete syntax

Whereas the abstract syntax specifies a set of legal abstract syntax trees, the *concrete syntax* serves as the basis of a parser that translates text into an abstract syntax tree. The concrete syntax specifications include the tokens and keywords that allow a parser to construct an abstract syntax tree unambiguously. In METAL, the concrete syntax part of a specification is distinct from the abstract syntax part. The abstract syntax is more "abstract" in that it only deals with types, not tokens, which may be altered (e.g., substituting the french token `lorsque` for the english token `when`) without changing the essential part of the syntax. Indeed, a given abstract syntax could have any number of associated concrete syntax specifications, although the METAL language does not currently allow for this.

METAL concrete syntax specifications are compiled into Yacc and Lex programs that are themselves compiled into a parser. METAL does not allow for the lexical specification of atomic values; this is done with Lex. The final parser runs as an independent process whose behavior is specified through resources.

4.2 The METAL environment

The METAL environment enables you to edit and compile METAL specifications. The METAL environment is available whenever an editor contains an abstract syntax tree belonging to the METAL formalism.

4.2.1 METAL tools

The METAL environment, in addition to pretty printing the METAL program, consists of an additional pulldown named *Metal* whose buttons allow us to type check, compile, and navigate through METAL programs.

You must first type check a METAL program for errors (since the compiler fails if the program contains any errors). If there are any errors or warnings, a special message window will open. You may click on any differently colored message text to see a corresponding position in the source.

To generate the tables for EXP, click on the *Compile Abstract* button of this menu. To compile both the abstract and concrete syntax parts of the program, click on the *Compile* button. This compilation does not create the final parser, which we discuss below.

4.3 The abstract syntax of EXP

We propose the following METAL syntax specification for EXP:

```
definition of Exp is
  abstract syntax
```

```
        exp_s -> EXP + ...;
        assign -> VAR EXP;
        plus -> EXP EXP;
        minus -> EXP EXP;
        prod -> EXP EXP;
        uminus -> EXP;
        variable -> implemented as IDENTIFIER;
        integer -> implemented as INTEGER;

        EXP_S ::= exp_s;
        EXP ::= assign plus minus prod uminus variable integer;
        VAR ::= variable;
        INTEGER ::= integer;
    end definition
```

The identifier following "definition of" is the name of the language.

The `exp_s` list operator must contain at least one expression. The `variable` and `integer` operators are atomic. Their implementation in Le-Lisp is described by the phrase `implemented as TYPE`. When an atomic value is recognized by the lexical analyzer, it is converted into the specified VTP type. This value is stored in the abstract syntax tree and may be pretty printed or retrieved by program. The remaining operators of the specification are fixed arity.

The `EXP_S` phylum contains the "root" operator of the language, `exp_s`. It does not appear on the right hand side of any operator definition, so the METAL type checker issues the warning `Unused Phylum`.[4] The `INTEGER` phylum does not appear on the right hand side of an operator definition, but we include for semantics purposes —it identifies a subset of expressions limited to integers. We refer to this phylum later in TYPOL programs.

Note that the `variable` operator belongs to two phyla. The `VAR` phylum allows us to restrict the type of operators considered valid on the left hand side of an assignment statement.

The abstract syntax defines valid sentences in the language that we represent as abstract syntax trees. For example, in figure 1 we show a legal tree that corresponds to the expression `B := A * (3 + 10)`. The definition of the `assign` operator prevents us from replacing the variable `B` with an

[4] Not including an operator in a phylum, on the other hand, does not prevent the construction of an abstract syntax tree but may pose problems during semantic analysis.

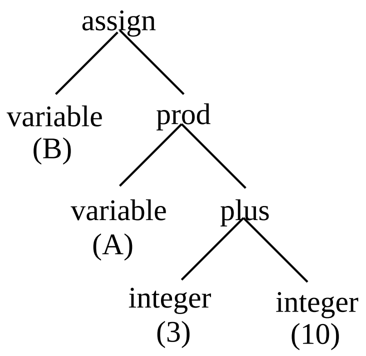

Figure 1: An abstract syntax tree.

operator that doesn't belong to the phylum `VAR`. Once we have an abstract syntax definition, we could manually construct abstract syntax trees using VTP primitives, but we prefer designing a parser to do the job.

4.3.1 Alternate abstract syntaxes

Let's look at another possible abstract syntax for EXP which imparts more type information and editing freedom. Instead of creating a new operator for each binary operator:

```
plus  -> EXP EXP ;
minus  -> EXP EXP ;
prod  -> EXP EXP ;
```

we may group all binary expressions under one operator with three tree descendents instead of two. The value of the atomic operator `binop` is a string that allows us to distinguish binary operators.

```
bexp -> EXP BINOP EXP ;
binop -> implemented as STRING ;
BINOP ::= binop ;
```

and change the phylum EXP accordingly:

```
EXP ::= bexp variable integer ... ;
```

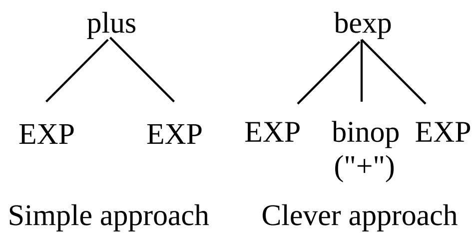

Figure 2: Simple versus clever abstract syntax trees

When we compare the small trees constructed with both approaches in Figure 2, several advantages of the "clever" approach stand out:

- Trees may be edited with more precision. For example, transforming `x + 3` into `x - 3` requires reanalyzing a much smaller subtree than would be the case with the simple approach.
- The grammar is more concise, and so pretty printers and semantic tools will be smaller and less prone to error.

The primary advantage of the simple approach is its simplicity, so we prefer sticking with it for the rest of the tutorial.

4.4 The concrete syntax of Exp

We complete the syntax for Exp by adding concrete syntax rules and tree building specifications for generating a parser. We design the concrete syntax according to the following specifications:

- Expressions must be *separated* by a semi-colon. We examine the difference between terminators and separators in the section on Ppml.
- We use the standard symbols to identify the corresponding operators: `+, -, *, :=`.

4.4.1 Production rules

A Metal rule consists of a *production rule*, or concrete syntax rule, and a *tree building function*. The parser identifies valid program sentences with these production rules. For instance, the production rule:

```
<exp> ::= <exp> "+" <exp> ;
```

tells the parser that two expressions separated by a + constitute a valid expression. All non-terminals appearing on the right hand side of a rule must themselves appear on the left hand side of at least one other rule. The names chosen for terminals and non-terminals need not be names from the abstract syntax definition, but we suggest using meaningful names (e.g., phyla names) whenever possible. Symbols and keywords to be recognized must appear within double quotes.

4.4.2 Tree construction

Once a production rule has recognized a stream of tokens, its tree building partner constructs a piece of the abstract syntax tree with the instantiated non-terminals and terminals that appear on the right hand side of the production rule. A tree must be constructed according to the arity of its associated operator: fixed, list, or atomic.

For example, in the case of addition, the tree to be constructed will have the associated fixed arity operator `plus`. This tree will have two descendents, the two arithmetic sub-expressions. To build the tree, we simple indicate the name of the associated operator and the list of descendent trees. The descendents are constructed during recursive phases of parsing. Thus, for addition, we have:

```
<exp> ::= <exp> "+" <exp> ;
   plus (<exp>.1, <exp>.2)
```

N.B. The "dotted" notation allows Metal to assign different values to the two <`exp`> variables, in the order of their appearance in the production rule.

To construct a tree with an associated list operator, the name of the operator is followed by a keyword that indicates how to construct the list. The keyword `list` means construct the list from scratch. The keywords `pre` and `post` mean add an element to the beginning or end of an existing list. For example, the first rule below:

```
<exp_s> ::= <exp> ;
   exp_s-list ((<exp>))
<exp_s> ::= <exp_s> ";" <exp> ;
   exp_s-post (<exp_s>, <exp>)
```

tells the parser to construct a tree whose operator is `exp_s` and whose list of descendents contains only one expression (note the two levels of parentheses to construct a list!). The second rule, when a list of expressions has been recognized, constructs the first part of the list recursively, then adds the last expression to the end of the list. Combined, these rules specify how to parse any list of expressions containing at least one expression and to construct a tree with these expressions as descendents. This corresponds to our abstract syntax specification the an `exp_s` list may not be empty.

To construct a tree with an associated atomic operator, the name of the operator in the tree building part is followed by the keyword `atom` and in parenthesis, a lexical variable name in capital letters. This variable is the name assigned within METAL to the atomic value as recognized by the lexer.

Consider the abstract syntax rule for an EXP variable:

```
variable -> implemented as IDENTIFIER ;
```

In the corresponding concrete syntax rule, we first tell the parser to recognize a token, which we name `%ID`. Then, we build a tree with the atomic operator `variable`. The atomic value for the operator is the `%ID` token.

```
<variable> ::= %ID ;
   variable-atom (%ID)
```

N.B. The symbols and keywords of the concrete syntax enable parsing but do not appear in the constructed abstract syntax tree.

4.4.3 Trivial rules and priority

The following rule does not construct a node in the abstract syntax tree:

```
<factor> ::= <unary> ;
   <unary>
```

but only passes along the result of subsequent instantiation of the non-terminal <`unary`>. The tree building part of the rule performs no additional work since the result of <`unary`> becomes the result of <`factor`>. This type of rule is frequently used to:

- ensure that a grammar is LALR(1) (since METAL depends on Yacc).
- establish operator priorities.

For example, production rules that treat all `EXP` operators at the same "level:"

```
<exp> ::= <exp> "+" <exp> ;
    plus (<exp>.1, <exp>.2)
<exp> ::= <exp> "-" <exp> ;
    minus (<exp>.1, <exp>.2)
<exp> ::= <exp> "*" <exp> ;
    prod (<exp>.1, <exp>.2)
```

not only ignore conventional priorities, but render parsing ambiguous for equations such as:

```
1 - 2 - 3
```

which, unrestricted by the above equations, may be left associative:

$$\boxed{\texttt{1 - 2}}\ \texttt{- 3}$$

or right associative:

$$\texttt{1 -}\ \boxed{\texttt{2 - 3}}$$

To remove this ambiguity, we create a trivial non-terminal named <factor> which ensures that the operator `prod` will be given priority over the `plus` and `minus` operators:

```
...
<exp> ::= <exp> "+" <factor> ;
    plus (<exp>, <factor>)
<exp> ::= <exp> "-" <factor> ;
    minus (<exp>, <factor>)
<exp> ::= <factor> ;
    <factor>
<factor> ::= <factor> "*" <unary> ;
    prod (<factor>, <unary>)
...
```

We will examine other priority issues, including the treatment of parentheses, in the section on pretty printing.

4.4.4 Metal and yacc

METAL is implemented using yacc and lex, so grammars you specify must belong to the class LALR(1). Making the parser involves compiling a Yacc program which may signal *shift-reduce* conflicts or *reduce-reduce* conflicts in the concrete syntax. These conflicts, not always harmful, should be resolved by making the concrete syntax LALR(1). You will find that the language EXP, as we have defined it above, contains two shift-reduce conflicts. They concern the production rules for `plus` and `minus`:

```
<exp> ::= <exp> "+" <factor> ;

<exp> ::= <exp> "-" <factor> ;
```

These conflict with the production rule concerning `assign`:

```
<exp> ::= <var> ":=" <exp>  ;
```

To picture why this is the case, suppose we have the expression:

```
a := b + c
```

We could parse it in two ways, by reducing the `assign` operator first:

$$\boxed{\texttt{a := b}}\ \texttt{+ c}$$

or, since in the abstract syntax the assignment statement is a valid expression (i.e., can be part of an addition), by shifting first, and reducing the `plus` operator before the `assign` operator:

$$\texttt{a :=}\ \boxed{\texttt{b + c}}$$

This is how we would rather interpret the expression. In the concrete syntax, we have not specified the precedences between the operators `assign` and `plus` and `minus`. By default, Yacc resolves the conflict by choosing the shift, but this treatment may not always be desirable or correct. Fortunately, in our case, this is what we want, so we can live with the conflict. To eliminate the conflicts we have two options:

- Add another token at the end of the `assign` production rule that allows Yacc to look ahead and resolve the conflict (the language is then LALR(1)). Unfortunately, this changes the concrete syntax of the language, which we don't want to do.

- Give the `assign` operator a higher priority by creating another non-terminal level (as we did for `prod`.)

To give `assign` a higher priority, we redefine the ambiguous part of the concrete syntax:

```
...
      <exp_s> ::= <assign> ;
         exp_s-list ((<assign>))
      <exp_s> ::= <exp_s> ";" <assign> ;
         exp_s-post (<exp_s>, <assign>)
      <assign> ::= <var> ":=" <exp>  ;
         assign (<var>, <exp>)
      <assign> ::= <exp> ;
         <exp>
      <exp> ::= <factor> ;
         <factor>
      <exp> ::= <exp> "+" <factor> ;
         plus (<exp>, <factor>)
      <exp> ::= <exp> "-" <factor> ;
         minus (<exp>, <factor>)
...
```

The parser generated from this definition requires us to add grouping parentheses for nested assignment statements:[5]

```
A := (B := 3 + 4)
```

4.4.5 The complete Exp specification

Here is the full definition of our language:

```
definition of Exp is
   rules
        <axiom> ::= <exp_s> ;
           <exp_s>
```

[5] To analyze the conflicts yourself, you might try applying Yacc by hand, with the "-v" (verbose) option to the `yaccExp.y` file generated by the METAL compiler. The resulting `y.output` file indicates the locations of the shift-reduce and reduce-reduce conflicts. Good luck.

```
      <exp_s> ::= <exp> ;
         exp_s-list ((<exp>))
      <exp_s> ::= <exp_s> ";" <exp> ;
         exp_s-post (<exp_s>,<exp>)
      <exp> ::= <variable> ":=" <exp> ;
         assign (<variable>,<exp>)
      <exp> ::= <factor> ;
         <factor>
      <exp> ::= <exp> "+" <factor> ;
         plus (<exp>,<factor>)
      <exp> ::= <exp> "-" <factor> ;
         minus (<exp>,<factor>)
      <factor> ::= <unary> ;
         <unary>
      <factor> ::= <factor> "*" <unary> ;
         prod (<factor>,<unary>)
      <unary> ::=  <term> ;
         <term>
      <unary> ::= "-" <term> ;
         uminus(<term>)
      <term> ::= <variable> ;
         <variable>
      <term> ::= <int> ;
         <int>
      <term> ::= "(" <exp> ")" ;
         <exp>
      <variable> ::= %ID ;
         variable-atom (%ID)
      <int> ::= %INTEGER ;
         integer-atom (%INTEGER)

abstract syntax
      exp_s -> EXP + ... ;
      assign -> VAR EXP ;
      plus -> EXP EXP ;
      minus -> EXP EXP ;
      prod -> EXP EXP ;
      uminus -> EXP ;
      variable -> implemented as IDENTIFIER ;
```

```
        integer -> implemented as INTEGER ;

        EXP_S ::= exp_s ;
        EXP ::= assign plus minus prod
               uminus variable integer ;
        VAR ::= variable ;

    rules
        <axiom> ::= "[EXP_S]" <exp_s> ;
           <exp_s>
        <axiom> ::= "[EXP]" <exp> ;
           <exp>
        <axiom> ::= "[VAR]" <variable> ;
           <variable>
  end definition
```

Normally the parser assumes that the very first production rule in the specification defines the beginning of a legal sentence. We may specify other *entry points* as well using the same left hand side as the first rule. This allows the parser to recognize program fragments instead of systematically trying to parse an entire program. The name of the first production rule is `<axiom>` which also appears on the left hand side of the last three rules so the parser will accept a list of expressions, a single expression, or just a variable as valid sentences. A special token appears on the right hand side of each `axiom` rule —the phylum name in brackets of the abstract syntax operator to be constructed. Since an operator may belong to different phyla, we specify the phyla that interests us explicitly, enabling the parser to construct the correct tree. Thus, if we want to parse just one expression instead of a list, we feed the line `[EXP] 89 * Myvar` to the parser.

4.5 An Exp parser

Having type checked the Exp program then compiled it (which generates a host of files in the directory specified by the resources), it is possible to build an Exp parser. To do so we must edit a few lines of files generated by the `centaur-language` command and the Metal compiler.

4.5.1 The Buildfile

The `Buildfile` contains directives for constructing the parser. Since this file was generated by the `centaur-language` command, we need not alter it. For details about Buildfiles, please consult the ctboot manual.

4.5.2 The lexical description

The METAL compiler creates a file, called `Exp.tokens.x` which provides the lexical description of EXP tokens. This file is used to construct a Lex scanner. It contains a list of regular expressions followed by actions. Usually, each action returns a token to the parser (via the **return** command). Returned tokens must bear the same names as the lexical variables that appear in atom production rules (in our case, `%ID` and `%INTEGER`).

The `Exp.token.x` file is merely a skeleton; it lacks the regular expressions that concern EXP. Thus, in the generated `Exp.tokens.x` file we find:

```
<sentences>ID                 return(ID);\
<sentences>INTEGER            return(INTEGER);\
```

We replace the token variable names by the desired regular expressions:

```
<sentences>[A-Z][A-Z0-9_]*   return(ID);\
<sentences>[0-9][0-9]*       return(INTEGER);\
```

The above regular expressions mean that EXP identifiers must begin with a capital letter followed by any combination and number of capital letters, digits, and underscores. Integers must be at least one digit long. Note that we handle the minus sign that appears before negative numbers in the `uminus` concrete syntax definition.

N.B. Make sure to save the modified tokens file as `Exp.tokens`.

4.5.3 Comments

The METAL compiler automatically generates abstract syntax operators to handle comments. But the language offers no mechanism to describe the concrete syntax of these comments. We must do this in the lexical description. For simple comment needs, such as one line comments, we only need to add a line to the tokens file indicating how the comment begins. For EXP, we specify that comments begin with a % symbol. In the `Exp.tokens` file, include one line with the appropriate regular expression and action just before the lines we previously modified. The result should be:

```
<sentences>"%"                 BEGIN comment;\
<sentences>[A-Z][A-Z0-9_]*     return(ID);\
<sentences>[0-9][0-9]*         return(INTEGER);\
```

When it encounters a %, the parser tells CENTAUR to store the text read until the end of the line in one of two predefined annotations named `prefix` and `postfix`.[6]

4.5.4 Parser creation and startup

To make the parser type `ctmake allmetal` in the directory containing the Buildfile. When necessary, CENTAUR will start up the new parser on the machine specified by the resources. If the parser is already running (from a previous CENTAUR session, for example), the old parser it used.

N.B. If you recompile the METAL specification of EXP, you must click on the *Load Syntax* button of the main menu's **Load** pulldown to kill the previous parser and start a new one.

4.5.5 Reading an Exp program

With a parser up and running, we may read an EXP program into a new ctview. We store example files in the directory `contrib/Exp/examples`. Resources determine what CENTAUR considers to be a valid suffix for a file containing a program in a given formalism. For EXP, we specify only one suffix, `.Exp`. This is specified in the file `contrib/Exp/syntax/Exp.rdb`.

Since we have not written a pretty printer for EXP, example trees are displayed using the default pretty printer, which shows the names and arities of operators and the values of atomic operators. A warning message in the CENTAUR main window tells us that the CENTAUR has not located the `Exp-std` pretty printer, which we define in the next section.

[6] The fact that the comment lasts until the end of the line is specified by preceding lines of this file, not shown here.

5 The Ppml definition

Now that we can parse an EXP program and read it into an editor view, we want to display it creatively and present important information through colors, fonts, selections, and annotations.

For all information about PPML, please consult the PPML manual in the Mantle.

5.1 Pretty printers

A pretty printer is made up of rules having the form:

pattern `->` *format*

where *pattern* indicates which abstract syntax operators a rule recognizes, and *format* describes their layout in terms of a *box language*. Pretty printer rules are compiled into code executed by the PPML *formatting machine*. In the format part of the rule, we may also assign resource names to tokens. The formatting machine pretty prints the tokens according to font and color resource values.

5.1.1 Modular pretty printers

PPML offers a *modular* pretty printing mechanism; a pretty printer may be composed of other pretty printers. A modular pretty printer has no associated PPML specification, it is simply specified by resources as a list of other pretty printers, which themselves may be modular or "real." You may dynamically and incrementally change the composition of a modular pretty printer in order to modify the presentation of a program. This is very important because it allows you to add to or change how a program is pretty printed without modifying existing programs.

We call the pretty printer from which the formatter has chosen a rule for the current subtree the *current* pretty printer.

5.1.2 Conventions

You may design any number pretty printers for the same language, but we propose the following convention:

- The `std` pretty printer should be the pretty printer "available to the public." It should be a modular pretty printer so that it may be modified either internally without requiring users to change their environments.

- The `basic` pretty printer should display an abstract syntax tree that, when saved in a file as seen on the screen (minus fonts and colors), may be reparsed directly. The `basic` pretty printer must exactly reproduce a concrete syntax specification (there may be several specifications). For example, the concrete syntax definition may dictate that a token separates expressions in a list or that it terminates each expression.

5.2 The formatting machine

The PPML formatter operates as follows. Given a compiled PPML specification and an abstract syntax tree, it traverses the tree, in search of a rule whose pattern matches the current tree node. The formatting machine observes the modular pretty printer mechanism, so it looks for a matching rule in the first pretty printer component. If it finds no appropriate rule, the second pretty printer in the list is consulted, and so on. The last pretty printer, which is always present and never fails, is the *generic* pretty printer, which displays the tree as we saw when we read an EXP program before having defined a pretty printer.

Once a rule has been found, the formatting machine resumes its tree traversal and consults the modular pretty printer's first component for a matching rule, etc. If any pretty printer components are themselves modular, the search for a rule continues recursively.

5.3 Patterns

A PPML rule's pattern contains textual and variable parts. A pattern matches a subtree when the textual part matches exactly and the variable parts may be instantiated (e.g., the signature of the tree's operator must match that of the pattern). The textual parts of the pattern are operator names from the associated abstract syntax specification. Variables represent descendents, atomic values, annotation names, print level, and other values.

Suppose we want to match the operator named `plus` which, as defined in the abstract syntax, has two descendents:

```
plus -> EXP EXP;
```

We would write the pattern:

```
plus(*x,*y)
```

where `*x` and `*y` are variables that will be instantiated farther down the tree as other expressions. The results of subsequent instantiations are used in the `format` part of the rule.[7] For `plus` we choose the format:

```
*x "+" *y
```

Variables that represent lists are prefixed by two asterisks, so that the pattern corresponding to a list of expressions defined in METAL as,

```
exp_s -> EXP + ...;
```

would be:

```
exp_s(**exp_list)
```

Variables in list arity operators may partition the list in a variety of ways, so if we want to extract the first expression in our list, for example, we would break it into a head and tail as follows:

```
exp_s(*head,**tail)
```

The operator `exp_s` still has one descendent, but since the descendent is a list, we may divide it up as we wish. The following pattern filters out the first and last elements of the list:

```
exp_s(*head,**body,*foot)
```

The following pattern, however, is **not** legal:

```
exp_s(**list1,**list2)
```

since PPML wouldn't know how to divide the list.

You may also match lists containing a specific number elements by using variables other than list variables. Thus, the pattern:

```
exp_s(*elem1,*elem2)
```

[7]Note that variable names are different since they are to be instantiated with different values. Variables may be referenced any number of times in the right hand part of a rule.

matches the list node `exp_s` when it contains only two elements. Do not confuse this with a fixed arity pattern with two descendents.

Variable names are only visible within a given rule and bear no relation to abstract operator names. We strongly suggest using meaningful names (such as phyla names) as mnemonic devices.

It is possible to have more than one pattern for the same abstract syntax node. For example, we may want to treat empty lists differently from non-empty ones:

```
exp_s()       -> ...

exp_s(**exp) -> ...
```

When the first of the above rules doesn't match (an empty list of expressions), the second is applied.

PPML applies rules in the order in which they are encountered. Thus, the pattern:

```
*anynode -> ...
```

which matches any node, should only appear at the end of a file, otherwise it would hide all subsequent rules. Such a rule is dangerous, even at the end of a program, because it matches everything, so you may not notice program bugs. Missing rules produce helpful warning messages during compilation, but no error since the generic pretty printer handles these nodes. In fact, the generic pretty printer allows you to develop a pretty printer incrementally since you may work on certain rules at a time, without worrying about the entire abstract syntax.

5.3.1 Print depth

Another pattern matching technique used frequently concerns the *print depth* of the tree we want to pretty print. The PPML environment allows us to specify how deep the pretty printer should descend when unparsing the tree. If, in the CENTAUR editor, we set the print level to 100, for example, the root of the tree will have a print level value of 100, its children 99, the grandchildren 98, etc.[8] We may want to reduce the print depth if, for

[8] By default, CENTAUR decreases the print level by one before unparsing a node's descendents. You may, however, change this behavior for each descendent explicitly in the right hand part of a rule.

example, unparsing the whole tree would produce a lot of text, and we want see the overall structure of the tree, without scrolling. The operator "`!`" in the pattern section of a rule, indicates at what print level a rule should be applied. Thus, the pattern:

```
*x !0 -> ...
```

matches any node when the print level is zero. The complete rule:

```
*x !0 -> [<h> "..."] ;
```

tells the pretty printer to print ellipses when the print level is zero. This saves a lot of space on the screen. Additionally, this saves some time, since no recursive unparsing calls are made after encountering a node whose print level is zero. Recursive calls in the formatting section of a rule are explained below.

For all pattern matching techniques, whether by text, print level, or annotation, remember that:

> *The most specific rules should appear at the beginning of the pretty printer specification.*

5.4 Formatting

The *format* part (right hand side) of a PPML rule expresses the concrete layout of patterns. Formatting directives may be any of the following:

- An empty box, used to format empty nodes.
- A box which may contain terminals, variables, and other boxes. Variables in a box represent recursive calls, and the results of instantiations of these variables eventually replace the variables at the same position.

  N.B. Recursive calls on atomic values that are not trees causes a fatal error during execution. The following rule will provoke such an error:

  ```
  id *x -> [<h> "$" *x] ;
  ```

The *type* of a box determines how each element— terminal, variable, or other boxes— will be aligned. PPML provides five types of boxes:

h : to concatenate elements horizontally.

v : to concatenate elements vertically. Elements after the first may be indented with respect to the left margin of the first.

v1 : like **v**, but elements after the first element may be indented with respect to the box surrounding the current box.

hv : to concatenate elements horizontally until the end of the line is reached. Boxes that don't fit on the current line are folded on to the following line(s) starting at the left margin of the current **hv** box.

hov : to concatenate elements horizontally until the end of the line is reached. If the elements don't all fit on the current line, the elements are all concatenated vertically.

The type of a box, also called the *global operator*, determines the spacing of all elements at the current level. It does not indicate how to format the elements found within nested boxes. In the following example:

```
[<v> *x *y [<h> *s "*" *t] *z]
```

the results of the instantiation of `*x`, `*y`, the inner <h> box and `*z` are aligned vertically, but `*s`, "`*`", and `*t` are aligned horizontally.

We format the `plus` operator as follows:

```
[<h> *x "+" *y]
```

PPML provides a device to treat lists —the parentheses operator. We may indicate, between parentheses, how to arrange tokens and elements of a list, and we may even use the "case" instruction to select formatting for different operators in the list. Suppose for our operator `exp_s` we simply want to line up all expressions vertically. We would write:

```
[<v> (**exp)]
```

The next box separates expressions horizontally, unless the list is too long, in which case it separates them vertically:

```
[<hov> (**exp)]
```

Note that the following construction:

```
[<v> (**exp ";")]
```

does not separate the elements vertically with a ";" at the end of each expression, but arranges everything vertically since the global operator <v> affects all elements in the box. The result is an expression on one line, followed by a semi-colon on the next line, the next expression on the following line, etc. To separate expressions by semi-colons, we modify the box to be:

```
[<hov> ([<h> **exps ";"]) *exp]
```

The <h> box within the parentheses overrides the global <hov> operator, so that each expression is paired with a semi-colon horizontally, except the final expression. These pairs are aligned horizontally as long as they all fit, otherwise they are aligned vertically.

Note that this rule corresponds to Exp's concrete syntax definition — expressions are separated by semi-colons. Another concrete syntax definition might have specified that each expression terminates with a semi-colon, in which case we would format each expression with:

```
[<h> *exp ";"]
```

and a list of expressions with:

```
[<hov> (**exp)]
```

In this case, the expression rule prints the semi-colon. If we send an expression to a textual editor, both the expression and the semi-colon appear in the editor. Since the concrete syntax definition requires that each expression be terminated by a semi-colon, we may immediately reparse the expression. Using this rule to pretty print Exp expressions would produce visually similar text, but pretty printed expressions would not be reparsable. The `basic` pretty printer should always format text so that it is reparsable.

In the above examples, we use default values for the different separators. It is often desirable to override these values locally. The Ppml reference manual explains the meaning of each separator's parameters. The <h> operator, for example, has one parameter indicating the number of blank spaces to appear between elements. For example, to specify that elements formatted by the `plus` rule should be separated by three spaces, we could write:

```
[<h 3> *x "+" *y]
```

We have everything we need for a simple pretty printer, though we will discuss other techniques and shortcuts presently.

5.5 The std pretty printer

As we mentioned above, the `std` pretty printer is a modular pretty printer. For the moment, its only component will be the `basic` pretty printer defined below. To specify the `std` pretty printer's composition, include the following line in the file:

`contrib/Exp/pprinters/pprinters.rdb`

```
Centaur.Exp.ppml.std.Package: (basic)
```

We will add to this list later on in the tutorial.

5.6 The basic pretty printer

We define the `basic` pretty printer will be defined in the file:

`contrib/Exp/pprinters/basic/Exp-basic.ppml`

The file name reflects the name of the formalism and the name of the pretty printer. Other files created by the PPML compiler will be generated in this same directory, according to resource specifications. A pretty printer should be viewed not as a handful of files, but as a module that is loaded into Le-Lisp and that may be compiled for efficiency reasons.

Here, once again, is the METAL abstract syntax of EXP:

```
abstract syntax
   exp_s  -> EXP + ... ;
   assign  -> VAR EXP ;
   plus  -> EXP EXP ;
   minus  -> EXP EXP ;
   prod  -> EXP EXP ;
   uminus -> EXP ;
   variable  -> implemented as IDENTIFIER ;
   integer  -> implemented as INTEGER;

   EXP_S  ::= exp_s ;
   EXP  ::= assign plus minus prod uminus variable integer ;
   VAR  ::= variable ;
   INTEGER  ::= integer ;
```

and here is a proposed first shot at the `basic` printer:

```
prettyprinter basic of Exp is

rules
  *x !0 -> [<h> "..."] ;

  exp_s(**exps,*exp)  -> [<v 0,0> ([<h 0> **exps ";"]) *exp] ;

  plus(*exp1, *exp2)  -> [<hv 1,2,0> *exp1  "+" *exp2] ;
  minus(*exp1, *exp2) -> [<hv 1,2,0> *exp1  "-" *exp2] ;
  prod(*exp1, *exp2)  -> [<hv 1,2,0> *exp1  "*" *exp2] ;

  uminus(*exp)        -> [<h 0> "-" *exp] ;

  assign(*var, *exp2) -> [<hv 1,2,0> *var ":=" *exp2] ;
end prettyprinter
```

Note that:

- the name of this pretty printer is `basic` for the language `Exp`.
- the first rule explains what to do for any node when the print level is zero. This rule is the most specific in our program, applying only when the print level is zero. Since we don't want it to be hidden by the other rules (which apply for any print level), it must be the first rule.
- we do not indicate how to display the terminals `variable` and `integer`. By default, the value of an atomic operator is pretty printed according to its type. You may also print atomic operators with predefined functions or by calling a user-defined external Le-Lisp function. This is explained later in the section on the `Exp-basic.at` file.
- the `exp_s` rule indicates that the semi-colon is a separator of expressions, not a terminator. The <h> box aligns each list element with a separator. The <v> box aligns these pairs vertically as well as the last list element. When the list contains only one element, the `**exps` list variable is empty, so we don't enter the parentheses at all.

5.6.1 Constants

Before compiling this program, we should make a few improvements. It would be helpful to localize separator information at the beginning of our program so that if we wanted to change spacing, we could just change one constant instead of lots of numbers. To declare constants in PPML, we use the optional *constant* section, which must be headed by the keyword `constant`. To replace the number **2** in our <hv> boxes with a constant which we call `tab`, we write:

```
prettyprinter basic of Exp is

constant
   tab = 2 ;

rules
   *x !0 -> [<h> "..."] ;

   exp_s(**exps,*exp)   -> [<v 0,0> ([<h 0> **exps ";"]) *exp] ;

   plus  (*exp1, *exp2) -> [<hv 1,tab,0> *exp1  "+"  *exp2] ;
   minus (*exp1, *exp2) -> [<hv 1,tab,0> *exp1  "-"  *exp2] ;
   prod  (*exp1, *exp2) -> [<hv 1,tab,0> *exp1  "*"  *exp2] ;

   uminus(*exp)         -> [<h 0> "-" *exp ] ;

   assign(*var,  *exp2) -> [<hv 1,tab,0> *var   ":=" *exp2] ;

end prettyprinter
```

5.6.2 Default parameters

Additionally, since most of our <hv> boxes use the same parameters, we might as well redefine the default values for <hv> in the optional default section so that we may use the abbreviated form of the separator. We leave the <h> box for `uminus` in place since we want the minus sign flush against the expression. The <h> and <v> boxes both have default values of zero. Our program becomes:

```
prettyprinter basic of Exp is
```

```
constant
   tab = 2 ;

default
   <hv 1,tab,0> ;

rules
   *x !0 -> [<h> "..."] ;

   exp_s(**exps,*exp)   -> [<v> ([<h> **exps ";"]) *exp] ;

   plus  (*exp1, *exp2) -> [<hv> *exp1  "+" *exp2] ;
   minus (*exp1, *exp2) -> [<hv> *exp1  "-" *exp2] ;
   prod  (*exp1, *exp2) -> [<hv> *exp1  "*" *exp2] ;

   uminus(*exp)         -> [<h> "-" *exp] ;

   assign(*var,  *exp2) -> [<hv> *var ":=" *exp2] ;

 end prettyprinter
```

5.7 Compilation

Before going any further, let's compile this version of the `basic` pretty printer to test its effect.

5.7.1 The Ppml program

The first phase of compilation involves compiling the Ppml program into a Le-Lisp module. This is done by clicking on the *Compile* button of the Ppml pulldown added by the Ppml environment. If your program contains any errors, a window opens up to display them.

Once the Ppml file has been compiled, we can display Exp trees with it. If an abstract tree for a program written in Exp exists already in another editor, selecting *Redraw* in the *Display* pulldown will have Centaur load the `basic` pretty printer and recompute the tree display. While developing a pretty printer, we recommend using two editors: one containing the pretty printer specification for your language and one containing an actual program written in the new language. You can thus very easily modify the Ppml file interactively, recompile it, and see the results.

5.7.2 The pretty printer module

To compile the module generated for the `basic` pretty printer, we rely on a `Buildfile` to direct the compilation from unix, as we did for Metal. We suppose that the Buildfile has been generated by `centaur-language`.

Typing `ctmake` in the directory containing `Exp-basic.ppml` compiles this file into a Le-Lisp module if necessary, then compiles the module. We suggest that you only compile a module once the pretty printer specification has stabilized. Use the interpreted version during development.

N.B.

- When you compile a Ppml program from Centaur, Centaur reloads the module automatically when the pretty printer is next used. The interpreted module is loaded if the pretty printer's *Mode* property value is **interp**. If a compiled module exists and the *Mode* property value is **compiled**, the compiled module is loaded.
- If you compile the generated module from unix, Centaur doesn't know about the modification. In this case, you must reload the pretty printer by hand by clicking on the *Load Pprinter* button of the main menu's **Load** pulldown. It is loaded according to the *Mode* property value.

Since we will continue to develop the pretty printer, we prefer to always load the interpreted module. Make sure the following resource appears in the file:

`contrib/Exp/pprinters/basic/basic.rdb`

```
Centaur.Exp.ppml.basic.Mode : interp
```

Don't forget to reset the resource manager by clicking on the *Reset Resources* button. When you are ready to used the compiled version of the pretty printer module, switch the value from `interp` to `compiled`.

Now we extend our pretty printer with more utilities offered by the Ppml language.

5.8 External formatting

CENTAUR provides a way to define Le-Lisp functions that may be referenced within the formatting part of a PPML rule. We store such functions in a file with the same name as the file containing the PPML rules, with the suffixe `.at`, which in our case would be `Exp-basic.at`. There are several cases where such functions are useful. The first concerns overriding PPML's automatic treatment of atomic operators. Patterns that recognize atomic operators have the general form:

```
atomic-operator-name *x -> ...
```

where `*x` receives the value of the atomic tree.

To treat our atomic operator `variable`, we might write the following rule:

```
variable *x -> [<h> var-handler(*x)]
```

The PPML formatter understands that `var-handler` is an external function call, so it looks in the `Exp-basic.at` for the function definition and executes it with the argument `*x`. Although we may choose any names we like for our functions, the package name must have the following form:

`#:pp`*language*__prettyprinter__`:atm:myfunction`

so that our `variable` handler must be named:

```
#:ppExpbasic:atm:var-handler
```

Since we only want our handler to return the value of `*x` as a string, the function is simply:

```
(de #:ppExpbasic:atm:var-handler (x)
    (string x))
```

We could, of course, perform other operations on `x`.

Another handy external function that we define allows us to associate tokens and operators. We have named this function `getsymbol`, which returns a string of characters for a given node name. For instance, we want to associate the token + with the operator `plus`. The function call:

```
getsymbol(plus)
```

within the formatting part of a rule should return the token +.

The function `getsymbol`, in combination with **where** allows us to do some fancy maneuvering. The **where** instruction allows us to limit operator matches to those operators found in a specific set. For example, the pattern:

```
*x where *x in {plus,minus,prod}
```

corresponds to any tree node whose operator matches one of the names within the braces. Once the variable `*x` has been matched, it may be used in the formatting section of the rule. Similarly, the pattern

```
*x where *x in {plus(*e1,*e2),minus(*e1,*e2)}
```

not only corresponds to the `plus` and `minus` nodes, but allows `*e1` and `*e2` to be instantiated so that they too may be used in the formatting section of the rule.

Instead of putting a set of operators to the right side of the word `in`, you can reference a phylum. Thus, the pattern:

```
*x where *x in EXP
```

means that `*x` can be any operator found in the phylum EXP as defined in our abstract syntax.

The following rule demonstrates how **where** and `getsymbol` may function together:

```
*x where *x in {plus(*e1,*e2),
                minus(*e1,*e2),
                prod(*e1,*e2)} ->

  [<hv> *e1 getsymbol(*x) *e2] ;
```

The variable `*x` in the PPML rule instantiates to the operator that matches the current node in the syntax tree. Of course we have to define what these tokens are in the function `getsymbol`:

```
(de #:ppExpbasic:atm:getsymbol (tree)
   (selectq ({tree}:operator-name tree)
      (plus "+")
      (prod "*")
      (minus "-")
      (uminus "-")
      (assign ":=")))
```

If you want to know more about the VTP functions that act on the `tree` argument, consult the VTP manual. All we need to remark here is that first argument of `selectq` is an operator name.

N.B.

- External functions must return a string or a Le-Lisp object on which one may call the function `string`.

- We strongly suggest using the **where** instruction, which compiles into smaller and more efficient Le-Lisp code. We will see later how to spot groups of similar rules that may be combined using **where**.

- In order to maintain *incremental* pretty printing, external function calls must be carefully managed. If an external function call on a tree T does not concern subtrees of T —as in our `getsymbol` function— you may tell the PPML formatter that the external function call will not affect incremental behavior. To do so, modify the external function definition as follows:

  ```
  (de #:ppExpbasic:atm:getsymbol (tree)
     (unless (eq 'evaluate #:decompilo:mode)
        (selectq ({tree}:operator-name tree)
           (plus "+")
           (prod "*")
           (minus "-")
           (uminus "-")
           (assign ":="))))
  ```

  Please consult the PPML user's manual for more information about optimizing incremental performance.

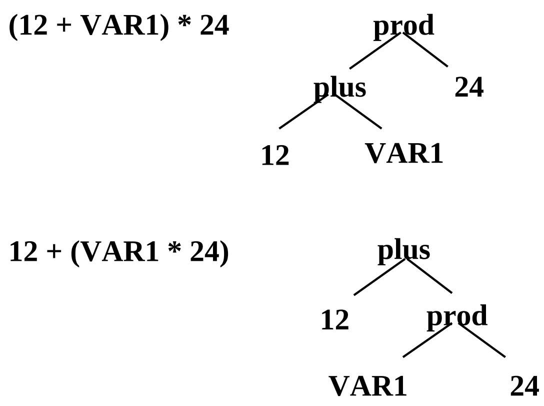

Figure 3: Determining priority.

5.9 More PPML tools: Restoring parentheses

Although straightforward, our preliminary pretty printer tells us nothing about operator priorities or parenthesized expressions. Our pretty printer formats the expressions `(12 + VAR1) * 24` and `12 + (VAR1 * 24)` identically, even though the underlying abstract trees differ. A better version of `basic` will insert clarifying parentheses. No operator of the abstract syntax corresponds to parentheses, so we cannot use simple pattern matching methods to recognize the operator hierarchy. To introduce priority, we make use of several sophisticated PPML features such as internal function definitions, the **if** control statement, the **where** statement, external functions calls, and contexts.

In both of the abstract syntax trees shown in figure 3, the expression in parentheses, which has the higher priority, appears farther down the tree. We know, therefore, that since the `plus` operator has a lower priority than the `prod` operator, when it appears farther down the tree it must be within parentheses. If we assign low integers to the operators with the highest priority (`*` = 2, `+`,`-` = 3, etc.), we can express our priority check as follows:

```
if {precedence(*left-child) > precedence (*current-op)}
   then put parentheses around *left-child
else
   no parentheses around *left-child
```

If the precedence of the right descendent is greater than or equal to the precedence of the current operator, we put parentheses around the right operator.

```
if {precedence(*right-child) > or = precedence (*current-op)}
   then put parentheses around *right-child
else
   no parentheses around *right-child
```

We treat the case of "equal to" since we want our expressions to be left associative.

5.9.1 Integer functions

In PPML, the **function** operator allows us to build functions that take abstract syntax trees as arguments and return an integer result according to the pattern of the tree's operator. We call our function `prec`, and define it before the **rules** section in the `Exp-basic.ppml` file:

```
function prec is
   prec (*op where *op in {assign}) = 4;
   prec (*op where *op in {plus,minus}) = 3;
   prec (*op where *op in {prod}) = 2;
   prec (*op where *op in {uminus}) = 1;
   prec (*op) = 0;
end prec ;
```

The last pattern catches all other operators and assigns them the highest priority. In EXP, these are `variable` and `integer`, which will never be surrounded by parentheses.

5.9.2 The if statement

The PPML **if** statement, which we can use to compare operator priority rating, has the following syntax:

```
if <condition1> then <format1>
[elsif <condition2> then <format2>]*
[else <format3>]
end if
```

where brackets indicate optional arguments and the asterisk indicates zero or more occurrences. The `<condition>` part may be:

- a boolean expression such as `*x in {plus,minus,prod}`
- a boolean expression using the operators `<`, `>`, `<=`, `>=`, `=`.

Combining `if` with the function `prec`, we produce the following rule for the `prod` operator:

```
prod(*exp1,*exp2)  ->

  [<hv> if prec(*exp1) > 2 then
           [<h> "(" *exp1 ")"]
        else
           *exp1
        end if
        "*"
        if prec(*exp2) >= 2 then
           [<h> "(" *exp2 ")"]
        else
           *exp2
        end if] ;
```

Rather than using the integer **2** explicitly, it would be better to recover the priority value of `prod` which we have already defined in the precedence function. Since we cannot write `prec(prod)` in the formatting part of the rule, we need to access the tree in the pattern part of the rule, using the **where** instruction. Our rule thus becomes:

```
*x where *x in {prod(*exp1,*exp2)}  ->

  [<hv> if prec(*exp1) > prec(*x) then
           [<h> "(" *exp1 ")"]
        else
           *exp1
        end if
        "*"
        if prec(*exp2) >= prec(*x) then
           [<h> "(" *exp2 ")"]
```

```
        else
           *exp2
        end if] ;
```

Since we know that this priority check is valid for any binary operator, we can use **where** to group the other operators into one general rule:

```
*x where *x in
  {plus(*exp1,*exp2), minus(*exp1,*exp2),
   prod(*exp1,*exp2), assign(*exp1,*exp2)} ->

  [<hv> if prec(*exp1) > prec(*x) then
           [<h> "(" *exp1 ")"]
        else
           *exp1
        end if

        getsymbol(*x)

        if prec(*exp2) >= prec(*x) then
           [<h> "(" *exp2 ")"]
        else
           *exp2
        end if] ;
```

Note that:

- we use the function `getsymbol` as described above to retrieve the token corresponding to the current operator.
- we include the `assign` operator even though its first descendent is in the phylum VAR (which is included in EXP!). The pretty printer doesn't care whether all of the operators in a **where** set have the same phyla. Knowing that an assignment statement has the same arity as the other operators and has a token between the descendents, we can include it in the same rule. We suggest scrutinizing your PPML programs for such similar rules and combining them using **where** constructs. This is not only more efficient, but by centralizing the information, you are more likely to find formatting errors and will have an easier time making changes in your program.

- if you parse a file with parentheses around an expression that already has priority, these parentheses will not be recreated by the pretty printer. Parentheses that provide concrete syntax information are lost in the abstract syntax tree.

5.9.3 Contexts: Improving selection behavior

Although we have successfully treated the case of parentheses, we still have one last problem to tackle: selection behavior. In a CENTAUR editor, when you click on a token, you select the tree node which generated this token during pretty printing, and all descendents of this node. Thus, in the following rule:

```
op(*op1,*op2) -> [<h> *op1 "*" [<v> "$" *op2]]
```

clicking on either `*` or `$` selects the `op` node and all its subtrees. On the other hand, clicking on either `*op1` or `*op2` means clicking on tokens that were unparsed farther down the tree, and thus the appropriate subtrees would be selected.

If we pretty print parentheses for the `prod` operator as follows:

```
*x where *x in {prod(*exp1,*exp2)}  ->

  [<hv> if prec(*exp1) > prec(*x) then
           [<h> "(" *exp1 ")"]
        else
           *exp1
        end if
        "*"
        if prec(*exp2) >= prec(*x) then
           [<h> "(" *exp2 ")"]
        else
           *exp2
        end if] ;
```

clicking on either parenthesis will always select the operator `prod` and its descendents:

```
(12 + var1) * 24
```

whereas the preferred response would be to select only the expression in parentheses, i.e., the `plus` operator:

$$\boxed{\texttt{(12 + var1)}}\ \texttt{* 24}$$

To remedy this behavior, we introduce a *context* into our program. A context is a subset of rules within a pretty printer specification. So far we have worked in the default context, but we may define others. Passing into a context restricts PPML's vision to rules whose patterns are prefaced by the context name. For example, the second of the following rules belongs to `mycontext`:

```
          op(*x) -> [<h> "hello" *x] ;
mycontext::op(*x) -> [<h> "goodbye" *x] ;
```

Any time we have passed into `mycontext`, the first rule is ignored. To pass into another context, we preface variable names (recursive calls) in the formatting section of a rule with the desired context name. Thus,

```
prod(*e1,*e2) ->
        [<h> mycontext::*e1  "*" mycontext::*e2] ;
```

passes into `mycontext` to treat the variables `*e1` and `*e2`.
When no rule is found in the current pretty printer for a matching a context, other pretty printers are examined in the order they are specified for the parent modular pretty printer.

How do we use contexts for EXP? We know at what moment we want to pretty print parentheses thanks to the precedence condition. We don't want to pretty print parentheses in this rule because of the mouse behavior, so we write another rule to print parentheses around any operator in the phylum EXP:

```
*x where *x in EXP -> [<h> "(" *x ")"] ;
```

This rule, however, would hide all other rules if at the beginning of a program, and would itself be hidden at the end of a program. If we define a context `paren`, we may pass into `paren`, apply this rule, and return to the original context to continue pretty printing. Passing into `paren` thus eliminates the problem of where to put the rule that prints parentheses. Thus, we apply the rule:

```
paren::  *x where *x in EXP -> [<h> "(" *x ")"] ;
```

any time we are certain that we want to print parentheses. Here is the full rule for the operator `prod` which relies on the above `paren` rule:

```
*x where *x in {prod(*exp1,*exp2)}  ->

  [<hv> if prec(*exp1) > prec(*x) then
           paren:: *exp1
        else
           *exp1
        end if
        "*"
        if prec(*exp2) >= prec(*x) then
           paren:: *exp2
        else
           *exp2
        end if] ;
```

Combining everything we've seen, our second version of `Exp-basic.ppml` is:

```
prettyprinter basic of Exp is

constant
   tab = 2;

default
   <hv 1,tab,0>;

function prec is
   prec (*op where *op in {assign}) = 4;
   prec (*op where *op in {plus,minus}) = 3;
   prec (*op where *op in {prod}) = 2;
   prec (*op where *op in {uminus}) = 1;
   prec (*op) = 0;
end prec ;

rules
   *x !0 -> [<h> "..."] ;
```

```
    exp_s(**exps,*exp) -> [<v> ([<h> **exps ";"]) *exp] ;

    paren:: *x where *x in EXP -> [<h> "(" *x ")"] ;

    *x where *x in {uminus(*exp)} ->
       [<hv> getsymbol(*x) if prec(*exp) > prec(*x) then
                              paren:: *exp
                           else
                              *exp
                           end if] ;

    *x where *x in
       {plus(*exp1,*exp2), prod(*exp1,*exp2),
        minus(*exp1,*exp2), assign(*exp1,*exp2)} ->
       [<hv> if prec(*exp1) > prec(*x) then
                paren:: *exp1
             else
                *exp1
             end if
             getsymbol(*x)
             if prec(*exp2) >= prec(*x) then
                paren:: *exp2
             else
                *exp2
             end if] ;
end prettyprinter
```

5.9.4 Comments : Pretty printing annotations

Parsers generated by METAL specifications store comments as annotations on the abstract syntax tree. A comment may be stored in one of two annotations, named `prefix` and `postfix`, depending on whether the comment should be pretty printed before or after the subtree to which it was attached.

The comment is stored as a tree whose associated (predefined) operator is `comment_s`, defined to be a list of comment lines. Each line is itself a subtree whose associated (predefined) atomic operator is named `comment`. Finally, the actual string of the comment is the value of this atomic operator. This implies that in order to pretty print a comment tree, we must add rules to our pretty printer to handle the `comment_s` and `comment` operators.

PPML's pattern matching mechanism allows you to recognize a tree annotation and filter the annotation value (like an atomic value). The following pattern matches the annotation named `myannot` on any subtree;

```
*x ^myannot -> ...
```

The annotation value is stored in the variable `^myannot` which may be used in the formatting part of the rule.

N.B. matching an annotation has the effect of rendering the annotation invisible for recursive calls, so that the pretty printer does not enter an infinite loop.

The following lines handle the comment annotations:

```
*x ^postfix -> [<hv> *x [<v> ^postfix ""]] ;
*x ^prefix -> [<v>  ^prefix *x] ;
```

The first line *re*formats the current tree (with the `postfix` annotation invisible), then the annotation value (tree whose operator is `comment_s`). The nested box has a vertical separator followed by the empty string at the end to ensure that the comment ends with a newline. The second line formats the annotation value, then reformats the current tree. These recursive calls on the same subtree are necessary because the comment annotation is independent of the subtree's arity.

Now we add two more lines to handle the annotation value: a comment subtree:

```
comment *x -> [<h> "%" stringpp(*x) ] ;
comment_s (**x) -> [<v> (**x) ] ;
```

The first line filters the comment string and reinserts the % symbol that was dropped during lexing. In this line, we explicitly handle the atomic value, bypassing the default mechanism. When an atomic value is not a subtree, we must tell PPML to stop recursion. We accomplish this by calling an external function with the value as argument. PPML offers a number of predefined functions to handle all sorts of values. In this example, `stringpp` simply returns the string value of the `comment` tree. The formatting machine requires an external function call to return a string or any value that responds to the Le-Lisp function `string`.

The second line treats a comment list by superposing lines.

5.10 Color and font specifications

And now the moment you've all been waiting for —including color and font specifications in a Ppml program. But first, a word about the Centaur resource manager, described in the resource manual of the Mantle.

5.10.1 The resource manager

Up to now we have avoided resources as much as possible. But now we must take a peak at the Centaur resource manager in order to add resources to our Ppml specification.

Each Centaur tool —pretty printer, formalism, graphical object, editor, etc.— has certain parameters, called *properties*, that the system or user should be able to specify and modify without altering the component's code. A tool may have subtools which may have their own subtools and so on, and at each level we should be able to specify property values. A *resource specification* is a pattern that traces a tool's position in the resource hierarchy (which is known in advance), designates one property for the tool, and furnishes a value. For example, to specify the background color of the `basic` pretty printer for the Exp formalism, we write:

```
*formalism-Exp.pprinter-basic.Format.Background: Blue
```

A resource pattern may consist of resource class names, instance names, question marks, and asterisks. A class name, such as `Format` above, means that all objects of this class will be given this resource value. An instance name, such as `Exp`, means that only instances of the `Formalism` class named `Exp` will be given this resource value. Thus, this specification does not apply to the `METAL` formalism, for example.

The question mark replaces one level of a resource pattern. The asterisk replaces any number of levels in the path. Thus, the line:

```
*Pprinter.Format.Background: Blue
```

specifies the background color for all pretty printers of all formalisms (the asterisk covers the formalism part). The line:

```
*Background: Blue
```

means that everything will have a blue background (selections, text, windows, etc.).

Normally we create a database file that contains a list of such specifications which the resource manager loads automatically into the current database. When a tool requires a property value, it provides a *query* to the resource manager who consults the current database for a pattern that matches (or returns a cached value).[9] However, the database may contain several patterns that apply for the same query. The most important thing to remember about resources is that when several patterns match, *the most specific resource pattern is always chosen.*

Precedence is defined in the following order, from most specific to least specific: instance name, class name, question mark, asterisk. Thus, if the current database contains these patterns:

```
*formalism-Exp*Background: Green
*formalism-Exp.Pprinter.Format.Background: Blue
*Pprinter.Format.Background: Yellow
```

and we request the background color for the `basic` pretty printer of the `Exp` formalism, the resource manager returns `Blue`. Note that between the two following resources:

```
Centaur*Background: Green
*formalism-Exp.Pprinter.Format.Background: Blue
```

the resource manager would return `Green` since at the first level, `Centaur` is more specific than the asterisk.

[9]The query is a list of instance names followed by a list of the same number of class names. It doesn't ever contain question marks or asterisks.

5.10.2 Ppml resource classes

The PPML language allows you to define a *resource class* for each token you pretty print. This class may then appear in a resource specification (after `Format`) followed by a `font`, `foreground`, or `background` property. For example let's reconsider the decision we made to let the default pretty printer handle variable names and numbers.

```
variable *x    -> [<h> stringpp(*x)];
integer *x -> [<h> integerpp(*x)];
```

The formatting machine expects a string or tree result from an external function call. Since the variable name is a string We add a class specification as follows:

```
variable *x    -> [<h> in class = variable : stringpp(*x)];
integer *x -> [<h> in class = number : integerpp(*x)];
```

which means "In a resource class named `variable` (or `number`), we pretty print the result of `stringpp`." When placed in front of a token or function call, `in class` affects the token only. When placed in front of a box, it affects every token or function call in the box as well as all tokens formatted by recursive calls. *Most recent* class specifications have priority, so specifications (for boxes, tokens, or function calls) inside a box preceded by `in class` override this box specification. Similarly, a class specification in a rule is superceded by class specifications in recursive calls.

Another form of class specification exists, `in term class`, that when placed in front of a box, affects every token or function call in the current box only. We use this construction to specify a resource class for comments:

```
comment *x ->
   [<h> in term class = comment : [<h> "%" stringpp(*x)]];
```

N.B.

- After adding the previous rules to the `Exp-basic.ppml` program, don't forget to *Compile* it.
- If no class has been specified for a token, either locally or by a surrounding box, font and color values are taken from the parent `Format` class.

5.10.3 Color and font resources

We may now add some color and font specifications to the file:

contrib/Exp/pprinters/basic/basic.rdb

```
!For variables
*visual-color*formalism-Exp.Pprinter.Format.class-name.foreground: red
*formalism-Exp.Pprinter.Format.class-name.font: \
-adobe-helvetica-*-r-*-*-14-100-*-*-*-82-*-*
!For integers
*visual-color*formalism-Exp.Pprinter.Format.class-number.foreground: RoyalBlue4
*formalism-Exp.Pprinter.Format.class-number.font : \
-adobe-new century schoolbook-bold-i-*-*-14-*-*-*-*-*-*-*
!For comments
*visual-color*formalism-Exp.Pprinter.Format.class-comment.background: misty rose
!For anything else.
*visual-color*formalism-Exp.Pprinter.Format.background: oldlace
*visual-color*formalism-Exp.Pprinter.Format.foreground: black
*formalism-Exp.Pprinter.Format.font: -*-lucida-medium-r-*-*-14-140-*-*-*-*-*-*
```

The first two specifications provide foreground and font values for the `name` class, the next two for the `number` class, and the last three specify the color and font of all other tokens (at the `Format` class level).

Notice that every resource pertains solely to the `basic` pretty printer. Resources common to all Exp pretty printers belong in the file:

contrib/Exp/pprinters/pprinters.rdb

Since the `basic.rdb` database file is only loaded when we use the `basic` pretty printer, resources that by mistake concern other pretty printers will not be visible before using the `basic` pretty printer. We avoid this kind of unhealthy dependency through the pretty printer manager.

The sole `comment` class resource specifies a background color for the comment text.

Note the use of the `Visual` class, instance `color`, which means that these resources only apply when using a color console. If you have a gray scale (instance `gray`) or monochromatic console (instance `mono`), you may want to specify other resources. Consider the case of the current selection. When we use Centaur with a color console we may present the current selection in living color, otherwise we may use a reverse video effect. Consider these specifications:

```
*formalism-Exp.Pprinter.Format.Select.foreground: none
*visual-color*formalism-Exp.Pprinter.Format.Select.Mode : transparent
*visual-color*formalism-Exp.Pprinter.Format.Select.background:LightSkyBlue1
*visual-mono*formalism-Exp.Pprinter.Format.Select.Mode : reverse
```

The first specification indicates that the foreground color of selected tokens will be `none`, a special value meaning "no value specified." Since we have not specified a foreground color, we may specify a `Mode` for the selection, indicating how to highlight the foreground of a selected token. The same principle applies to the selection background, but here we will only examine the foreground presentation. The value `transparent` means that the static token color will be used. Thus, numbers will appear in `RoyalBlue4` and variables in `red` even when selected. However, we specify the selection background as `LightSkyBlue1`, so the tokens will be surrounded by light blue. Instead of `transparent` we could have chosen other modes described in the resource manual. The last specification means that using a monochrome console, the selection is displayed in reverse video.

Other than color and font resources, we may specify the default print level of a program, the width of a line, the distance between lines, etc.

```
*formalism-Exp.Pprinter.Format.vskip : 1
*formalism-Exp.Pprinter.width: 80
*formalism-Exp.Pprinter.printLevel: 10
```

To test these resources, click on *Reset Resources* and then reread a file or click on the *Redraw* button of an existing ctview containing an Exp program.

N.B. In the previous resources, we have used the class name *Pprinter* everywhere instead of the instance name `pprinter-basic`. We intend to work with the modular pretty printer `std` instead of `basic` directly. If the pretty printer's name is `std`, resources with the instance name `basic` will not be considered. Using the class name *Pprinter* (or the question mark) allows these resources to be visible for any Exp pretty printer, including `std`.

6 The Typol specification

We now go beyond the syntactic realm to describe some semantics of the EXP language with a set of TYPOL rules. We might want to study many different semantics when developing a language —static, dynamic, transformation, etc. In this tutorial, we specify the dynamic semantics of evaluating arithmetic expressions. We compile the TYPOL program that specifies these semantics into an interpreter for EXP.

Thus far, the EXP environment consists of a parser tool and a pretty printer tool. We are about to add an interpreter tool that is implemented in TYPOL. A semantics tool may require more than one TYPOL program to function; these files are all stored in the directory specified by resources in the formalism resource file. The TYPOL specifications for the EXP interpreter are stored in the directory:

```
contrib/Exp/semantics/interpreter
```

For information about TYPOL, please consult the TYPOL manual in the Mantle. For information about messages, please consult the message manual in the Core.

6.1 Typol and Prolog

The current version of TYPOL is implemented in Prolog, and understanding how Prolog works will certainly make grasping TYPOL easier. CENTAUR uses TYPOL to define semantics rather than Prolog, however, for several reasons:

- TYPOL offers a general way of expressing semantic rules, independent of the *target language*, i.e., the language into which it is compiled (here, Prolog).
- TYPOL offers advanced interactive tracing mechanisms.
- TYPOL offers automatic type checking of instantiated variables, based on the language formalism definition.

TYPOL, like Prolog, manipulates abstract syntax tree fragments as terms, as in the expression:

```
plus(prod(integer 1, integer 2), integer 3)
```

6.2 Typol and natural semantics

In natural semantics, computing the value of an expression in a language amounts to proving that a particular proposition holds. A proposition holds when it can be formally derived from the set of rules that constitute the TYPOL program. Thus, the proposition:

```
X is half a man.
```

would hold if we had expressed the fact that:

CENTAUR `is half a man.`

somewhere in a set of TYPOL rules. The symbol `X` is a variable. For the expression to hold, `X` must:

- be free, i.e., have no value associated, in which case it is instantiated to the value CENTAUR.
- already be bound to the value CENTAUR.

A proposition in TYPOL has the syntax:

```
<expression_s> [ <symbol> <expression_s> ]
```

where `<expression_s>` is a list of expressions separated by commas. These expressions are terms. They are patterns that represent fragments of an abstract syntax tree. These patterns contain operator names from the formalism and may contain metavariables that stand for subtrees. An atomic tree pattern is the name of the atomic operator followed by an appropriate value.

The `<symbol>` and second `<expression_s>` are optional. The symbols that appear in semantic equations have no intrinsic meaning, they only separate TYPOL identifiers (variables, syntactic operators, phyla, programs, types, ...). However, the turnstyle ($\vdash$) has a conventional meaning that it retains in TYPOL.

To express the above proposition in TYPOL, we might write:

```
X : half a man.
```

which would hold given a set of rules that included:

CENTAUR : `half a man.`

In EXP, evaluating an arithmetic expression amounts to proving that the expression may be reduced to a single integer value. An EXP proposition should express the relationship between the original expression and the final integer value. In English, the proposition might be "the value of expression `exp` is `val`." In TYPOL, we have adopted the notation `exp : val`.

6.2.1 Sequents

The turnstyle must appear in every *sequent.*[10] A sequent is a proposition combined with the hypotheses necessary to prove it. A TYPOL sequent has the syntax:

```
<hypotheses> "|-" <consequent>
```

where `<hypotheses>` is a list of expressions separated by commas, and `<consequent>` is a proposition.

Since we don't require any hypotheses when evaluating pure arithmetic expressions —the evaluation of expressions is context independent— our sequents initially have the form:

$$\vdash \texttt{expression : value}$$

The very first abstract syntax expression in the consequent of a sequent is called the **subject** of the sequent. The subject has a particular role during the proof of a propostion, as we will see shortly.

6.2.2 Judgements

In TYPOL, sequents are rigorously typed and must be declared at the beginning of a program by a *judgement* declaration. The judgement declares a syntactic form that may be used throughout the program. The form consists of expressions and symbols. The expressions may be any of the following:

- The name of a phylum from one of the formalisms **use**d at by the program. All formalisms used in a program must be declared in a *use* clause at the beginning of the program.

[10]Not exactly. If the sequent has no hypotheses and its consequent is a simple list of expressions, you may omit the turnstyle and put the consequent in parentheses.

- A formalism name, which stands for all of its phyla.
- The predefined Typol types `integer`, `string`, and `path`.
- The anonymous variable underscore (_), which corresponds to any type.

For example, the judgement:

$$\vdash \texttt{EXP : INTEGER}$$

declares that expression to the left of the colon must be an abstract syntax tree pattern whose associated operator belongs to the phylum `EXP`. To the right of the colon, the pattern's associated operator must belong to the phylum `INTEGER`.

The profile of the judgement —the number and syntax of symbols and the number of elements in the hypotheses and consequents— may only appear once in a program. For example, the following judgements could not both appear in the same program:[11]

```
judgement ENV |- DECLS -> ENV
judgement ENV |- RULES -> ENV
```

A sequent that doesn't respect the abstract syntax constraints imposed by the judgement is rejected by the Typol type checker.

What judgement(s) should we declare for Exp? The most natural response is:

$$\texttt{judgement} \vdash \texttt{EXP : INTEGER}$$

However, we want to evaluate not only an expression, but a list of expressions (phylum `EXP_S`). We can either add a second judgement to handle an expression list, create a larger phylum to englobe both `EXP` and `EXP_S`, or use the language name to allow all Exp phlya. We choose the later solution for the sake of simplicity, but we discourage this choice in general because it allows overly permissive, weakly typed semantic rules.

To the right of the colon, we specify `INTEGER`, which contains the operator `integer`, the lone operator in Exp that expresses a value. Our judgement declaration therefore has the form:

$$\texttt{judgement} \vdash \texttt{Exp : INTEGER}$$

[11] To resolve this conflict, you could either replace the phyla `RULES` and `DECLS` by a phylum containing the two, or replace `->` by a different symol.

6.2.3 Rules

A TYPOL program consists of a list of global declarations (judgements, import clauses, definitions) and a set, or sets, of *rules* which may be either *inference rules* or *axioms*. These rules explain how to deduce sequents from other sequents.

Inference rules have the following form:

$$\frac{premises}{conclusion}$$

TYPOL semantics state that when we have proved all of the premises, we have proved the conclusion and thus the rule. The numerator is a list of sequents, the conclusion a single sequent, and the subject of the conclusion is called the subject of the rule. The inference rule:

$$\frac{\vdash\ exp_1\ :\ integer\ x_1 \qquad \vdash\ exp_2\ :\ integer\ x_2 \qquad add(x_1, x_2, x)}{\vdash\ plus(exp_1, exp_2)\ :\ integer\ x}$$

may be interpreted as follows:

> *The value of the tree (whose associated operator is* `plus`*) with descendents exp_1 and exp_2 is x where x_1 is the result of the evaluation of exp_1, x_2 is the result of the evaluation of exp_2, and x is the result of the addition of x_1 and x_2.*

This rule has three premises, two of which are recursive calls to evaluate the descendents of the subject tree, and the third actually adds the resulting integers. The subject of this rule is the pattern $plus(exp_1, exp_2)$.

Axioms are rules without premises and therefore serve to stop recursive deduction. In EXP, we use the axiom:

$$\vdash\ integer\ N\ :\ integer\ N$$

which may be interpreted as *The value of the integer N is itself.* The word `integer` that appears in the axiom and previous inference rule is the name of the atomic operator in the abstract syntax.

6.2.4 The Typol motor

How do we prove a proposition given a set of rules? The prolog implementation of the TYPOL motor takes a set of rules, initial data, and a proposition.

Some of the free variables of the proposition are instantiated with the initial data. Input data may be integers, strings, and abstract syntax trees. The motor then attempts to prove the proposition within the set of rules, instantiating the remaining free variables along the way. When the proposition may be proved, we usually retrieve the values of the newly instantiated variables

The initial value assigned to the subject of the proposition is usually the root of an abstract syntax tree. The TYPOL motor tries to prove the proposition by finding a rule whose subject pattern matches the proposition's subject tree. When a match occurs, the matching rule's hypotheses are instantiated to those of the proposition. Free variables of the rule's patterns are instantiated to corresponding subtrees in the proposition.

Once the motor finds a rule that matches a proposition, it attempts to prove recursively the premises of the rules. If the rule is an axiom, no further proof is necessary.

If, while the motor is trying to prove a rule, a premise fails, the rule does not prove the proposition. So the motor looks for another matching rule. In TYPOL, a proposition may not be proved if the motor finds no matching rules.

For example, we may ask the TYPOL motor to prove the following proposition:

```
|- Exp : integer N
```

given a set of rules. The variable `Exp` is bound to a source tree. The variable `N` is initially free. If the proposition may be proved, `N` will contain the single integer result of the evaluation.

6.3 A simple interpreter

For simplicity's sake, we limit the scope of our first interpreter to the addition, subtraction, and multiplication of integers in a list of expressions. Here is the specification for our interpreter in words:

- The value of an integer i is itself.
- The value of a binary expression is the result of a binary operator applied to the evaluation of the two descendents.
- The value of a unary expression is the result of a unary operator applied to the evaluation of its descendent.

- The value returned by a list of expressions is the value of the last expression.

We express these *rules*, using the notation of natural semantics, as follows:

$$\vdash \ integer\ N : integer\ N$$

$$\frac{\vdash \ exp_1 : integer\ x_1 \qquad \vdash \ exp_2 : integer\ x_2 \qquad add(x_1, x_2, x)}{\vdash \ plus(exp_1, exp_2) : integer\ x}$$

$$\frac{\vdash \ exp_1 : integer\ x_1 \qquad \vdash \ exp_2 : integer\ x_2 \qquad sub(x_1, x_2, x)}{\vdash \ minus(exp_1, exp_2) : integer\ x}$$

$$\frac{\vdash \ exp_1 : integer\ x_1 \qquad \vdash \ exp_2 : integer\ x_2 \qquad mult(x_1, x_2, x)}{\vdash \ prod(exp_1, exp_2) : integer\ x}$$

$$\frac{\vdash \ exp : integer\ x_1 \qquad neg(x_1, x)}{\vdash \ uminus(exp) : integer\ x}$$

$$\frac{\vdash \ exp_1 : v}{\vdash \ exp\_s[exp_1] : v}$$

$$\frac{\vdash \ exp : v1 \qquad \vdash \ exps : v}{\vdash \ exp\_s[exp.exps] : v}$$

The last two rules correspond to the fourth item in our specification and may be read:

> *If a list of expressions contains one expression, the value returned is the result of its evaluation. If a list contains more than one expression, evaluate the first expression and return the result of the evaluation of the rest of the list.*

Using the true TYPOL syntax, we would write these rules as follows:

```
|- integer N : integer N ;

|- EXP1 : integer x1  &
|- EXP2 : integer x2  &
add(x1, x2, x)
----------------
|- plus(EXP1, EXP2) : integer x ;

|- EXP1 : integer x1  &
|- EXP2 : integer x2  &
sub(x1, x2, x)
----------------
|- minus(EXP1, EXP2) : integer x ;

|- EXP1 : integer x1  &
|- EXP2 : integer x2  &
mult(x1, x2, x)
----------------
|- prod(EXP1, EXP2) : integer x ;

|- EXP  : integer x1  &
neg(x1, x)
----------------
|- uminus(EXP) : integer x ;

|- EXP1 : v
----------------
|- exp_s[EXP1] : v ;

|- EXP1 : v1  &
|- EXPS : v
----------------
|- exp_s[EXP1.EXPS] : v ;
```

Lists in TYPOL are designated using brackets, and list elements are distinguished using the period (.) notation.[12]

These rules do not yet compose a complete TYPOL program, but they make up the purely semantic part. The rest of the program will address

[12] TYPOL keywords must be prefixed by a sharp sign ("#").

input and output issues.

6.4 Communication between Typol and LeLisp

How does Typol assign initial values to a proposition? And how do we recover the result(s) of the proof? In almost all circumstances, abstract syntax trees that are initial values of propositions reside in Le-Lisp and must be sent to prolog, being coerced into a prolog term along the way. Results send by prolog are similarly coerced into Le-Lisp representations.

6.4.1 Typol side

The first step to establishing communication between a Typol program and the Le-Lisp environment is to *export* a proposition that you wish to make, i.e., trigger from the Le-Lisp world. The Typol compiler generates Le-Lisp function associated with every exported Typol proposition. The function's arguments are the initial values of the proposition. The function takes care of coercing these values to prolog terms.

The `export` command has the syntax:

```
export <premises> as
       <function-name> (<val1>, <val2>, ...) = <result> ;
```

where `<function-name>` will be the (unpackaged) name of the generated function, `<val1>, <val2>, ...` its arguments, and `<result>` its result. The arguments and result correspond to variable names in the premises of `<premises>`, which is an axiom or the denominator of a rule. It is also possible to return several results in a list, using the syntax:

```
export <premises> as
       <function-name> (<val1>, <val2>, ...) =
                       (<result1>, <result2>, ...) ;
```

Since our interrogations in Exp have the form:

```
|- exp : integer N;
```

our `export` clause is:

```
export |- exp : integer N as eval_exp(exp) = N;
```

The generated function `eval_exp` takes one argument, a tree, which will become the value of the TYPOL variable `exp` before TYPOL tries to prove the proposition `|- exp :  integer N`. The result of the proof, named `N` in TYPOL, will be returned by the function `eval_exp`.

The above description is only partially accurate because the name of any Le-Lisp function generated by the compilation of an export statement has the following syntax:

```
#:program_name:function_name
```

so that in our example, the full generated function name would be `#:eval_exp:eval_exp`.

N.B. You should not call a function generated by an `export` clause directly but use either the `ty-synchronous-run` function for batch mode executions (for example, to quickly test a TYPOL program) or an interactive environment as described in section 7.

6.5 Importing prolog predicates

We have just a couple of loose ends to tie up before finishing our program. Since we have used several external functions in our TYPOL rules (add, sub, mult, and neg), we must herald their origins using the *import* clause:

`import` *function1(arg1,arg2), function2(arg1),...* `from` *program*;

When importing a predicate, you must indicate the correct number of arguments, with either an underscore or type name indicating the type of each argument. For our arithmetic predicates, we write:

```
import add(integer, integer, integer),
       sub(integer, integer, integer),
       mult(integer, integer, integer),
       neg(integer,integer) from expfunc;
```

where we have chosen `expfunc` as a package name for functions found in a prolog file named `expfunc.sp`. Any predicate `function` imported from `package` must be named `package$predicate` and be defined in a file `package.sp`. The package name must begin with a lower case letter since it is a prolog predicate. Our file `expfunc.sp` contains the following definitions:

```
expfunc$add(N, M, X)  :- X is N + M.
expfunc$sum(N, M, X)  :- X is N - M.
expfunc$mult(N, M, X) :- X is N * M.
expfunc$neg(N, M)     :- M is 0 - N.
expfunc$diff(N, M)    :- N ~= M.
```

We have now seen the minimum necessary for building a simple interpreter for Exp. The following program should be stored in the file:

`contrib/Exp/semantics/interpreter/eval_exp.ty`

```
program eval_exp is
    use Exp;
    import add(integer, integer, integer),
           sub(integer, integer, integer),
           mult(integer, integer, integer),
           neg(integer,integer) from expfunc ;

    export |- EXP : integer N as eval_exp(EXP) = N ;

    judgement |- Exp : INTEGER ;


    |- integer N : integer N ;

    |- EXP1 : integer x1  &
    |- EXP2 : integer x2  &
    add(x1, x2, x)
    ----------------
    |- plus(EXP1, EXP2) : integer x ;

    |- EXP1 : integer x1  &
    |- EXP2 : integer x2  &
    sub(x1, x2, x)
    ----------------
    |- minus(EXP1, EXP2) : integer x ;

    |- EXP1 : integer x1  &
    |- EXP2 : integer x2  &
```

```
    mult(x1, x2, x)
    ----------------
    |- prod(EXP1, EXP2) : integer x ;

    |- EXP : integer x1  &
    neg(x1, x)
    ----------------
    |- uminus(EXP) : integer x ;

    |- EXP1 : v
    ----------------
    |- exp_s[EXP1] : v ;

    |- EXP1 : v1  &
    |- EXPS : v
    ----------------
    |- exp_s[EXP1.EXPS] : v ;

end eval_exp ;
```

6.6 The Typol programming environment

The Typol programming environment consists of two type checkers, a compiler, code generator, and debugger. Aside from the debugger, the tools may be called by clicking on buttons in the popup menu associated with the right mouse button in editors containing Typol programs, or in the Typol pulldown menu. You must always type check (button *Type Check*), compile, and generate code for a given program, in that order. Generated files are stored in the same directory that contains the source.

You may interrupt compilation or type checking by clicking in the *Abort* item of the popup menu which is temporarily associated with the right mouse button during the three phases of compilation. Since the prolog process runs asynchronously, Centaur responds to input during type checking, etc.

6.6.1 Automatic loading

Typol automatically loads the results of the compilation into the prolog and Le-Lisp environments when necessary by using resources to locate the appropriate files (see the formalism resource file).

6.6.2 Debugging a Typol program

In case the execution of the program fails, you may want to recompile the Typol program in *debug mode*, by clicking on the $\boxed{\textit{Compile(Db)}}$ item of the $\boxed{\textit{Typol}}$ pulldown, followed once more by $\boxed{\textit{Write Code}}$. Any time you recompile a Typol program, you may reload it into the prolog environment using the $\boxed{\textit{Reload Code}}$ button. You may clear the entire prolog environment by clicking on the $\boxed{\textit{Restore Prolog}}$ button.

When you run a program that has been compiled in debug mode, the debugger interface opens automatically. The Typol users manual provides a complete debugging example.

6.6.3 Batch compilation

It is also possible to Typol programs in *batch mode*, i.e., without the graphical user interface. To do so, the `Buildfile` generated by the centaur-language command must be completed with target file information. Please consult the Typol users manual for details.

6.7 A better interpreter for Exp

Now we want to add the treatment of variables and the assignment statement to our interpreter. Adding variables poses a challenge because we need to be able to remember their values from one expression to the next. We can no longer evaluate expressions without concern for the evaluation *environment*, or context. In our case, the environment is simply a set of variables and their values. To correctly account for the assignment expression, a proposition must be proved in a context. And, the proof of the proposition may alter the context. Each context must also be propagated to premises of rules used to prove the proposition.

With this in mind, we introduce additional semantic specifications to handle variables and the assignment statement:

To evaluate the `variable` operator:

- If the variable is not yet in the environment, add it with the initial value 0 and return 0 and the new environment.
- If the variable is in the environment, return its value and the unchanged environment.

To evaluate the `assign` operator:

- If the variable being affected is not yet in the environment, add it to the environment with the new value and return this new value and modified environment.
- If the variable being affected is in the environment, change its value in the environment and return the new value and modified environment.

6.7.1 The environment

We may think of an environment as a hypothesis in a proposition: "Given a certain environment, evaluate an expression." In sequents, hypotheses appear to the left of the turnstyle. Since the result of the evaluation of a premise may alter the environment, we return the modified environment as well as the result of the evaluation. Here is the modified Exp judgement:

$$\texttt{ENV} \vdash \texttt{Exp : INTEGER, ENV}$$

Note that since we have changed the judgement, we must modify the syntax of all previously defined rules, adding the initial and final environments.

The phylum `ENV` stands for the environment. Our environment is a list of pairs of the form **variable**,**value**. Since the environment is manipulated by a Typol program, it should be implemented as an abstract syntax tree. We may either implement the environment tree in a distinct formalism specification, dedicated to this task, or augment the Exp language to add operators for the environment. For the sake of simplicity, we prefer to modify the Exp abstract syntax as follows:

```
env -> PAIR * ... ;
pair -> VAR INTEGER ;

ENV  ::= env ;
PAIR ::= pair ;
```

We do not add concrete syntax rules for these operators since we don't intend to parse the environment. Where do environment trees come from?

- Typol: Typol patterns will be used to build the environment during evaluation. These patterns will be coerced into Vtp trees automatically when sent to Le-Lisp.

- *Le-Lisp*: We may either construct VTP trees by hand, (using low level VTP primitives) or through a graphical interface. We may use generic cut and paste operations to build an environment tree with no modifications of the METAL program. However, in order to use the guided editing mode or in-line edition, CENTAUR must be able to send integers and variables to an EXP parser. We must therefore add the following entry point to the METAL definition to parse integers:

  ```
  <axiom> ::= "[INTEGER]" <int> ;
        <int>
  ```

  This line may accompany the previous entry point definitions in the specification.

Please add the above lines to `contrib/Exp/syntax/Exp.metal`, recompile it, regenerate the parser, then reset the syntax from the main CENTAUR menu.

6.7.2 Pretty printing the environment tree

In section 7, we will explain how to display an EXP environment in a separate view. To pretty print the environment tree in this view, we add a specialized pretty printer named `env` to our modular pretty printer `std`.

First, add the name `env` to the *Package* resource for the `std` pretty printer in
`contrib/Exp/pprinters/pprinters.rdb`:

```
Centaur.Exp.ppml.std.Package: (env basic)
```

Then define the `env` pretty printer in `contrib/Exp/pprinters/env/Exp-env.ppml` to be:

```
prettyprinter env of Exp is
  rules
      env(**pair) -> [<v> (**pair)] ;

      pair(*variable, *integer) ->
          [<hv 1,3,0> env::*variable "=" env::*integer] ;
```

```
        env::integer *x ->
             [<h> in class = env-number : integerpp(*x)] ;
        env::variable 'RESULT' ->
             [<h> in class = env-result : "RESULT"] ;
        env::variable *x ->
             [<h> in class = env-name : stringpp(*x)] ;
  end prettyprinter
```

The program formats the environment as a vertical series of pairs having the form `identifier = value`. Note the use of the `env` PPML context. When the `pair` operator is matched, we know that we are in the midst of pretty printing the environment. We prefix the recursive calls on the variable and integer subtrees with the `env` context so that normal rules for printing these operators will be hidden and only the special environment rules will match. If we had not used a context, during the recursive call we would have lost the information about being in an environment.

The three rules preceded by the `env` context format the variable and integer of each pair. The `RESULT` is treated separately thanks to pattern matching the name of the variable. These rules actually only serve to define special resource classes for special fonts and colors; we could have otherwise used the basic `integer` and `variable` rules.

Please add the corresponding resources to `contrib/Exp/pprinters/env/env.rdb`:

```
!Colors
*visual-color*formalism-Exp.Pprinter.Format.class-env-name.foreground: black
*visual-color*formalism-Exp.Pprinter.Format.class-env-number.foreground: \
RoyalBlue4
*visual-color*formalism-Exp.Pprinter.Format.class-env-result.foreground: red
!Fonts
*formalism-Exp.Pprinter.Format.class-env-name.font: \
-adobe-helvetica-*-r-*-*-14-100-*-*-*-82-*-*
*formalism-Exp.Pprinter.Format.class-env-number.font : \
-b&h-*-bold-r-*-*-12-*-*-*-*-79-*-*
*formalism-Exp.Pprinter.Format.class-env-result.font: \
-*-new century schoolbook-*-i-*-*-17-*-*-*-*-*-*-*
```

And of course, reset the database from the main menu.

6.7.3 Evaluating in an environment

To underline the fact that the semantics concerning the environment differ from those of pure arithmetic, we will describe them in distinct TYPOL *sets*. A set is a named collection of declarations and inference rules. A *program* such as `eval_exp` is just a special kind of set. Sequents used within a set refer to other rules within the set unless preceded by the name of a different set. Rules in one set may be used in another, which allows you to define the semantics of a language in a modular fashion.

A set may have declarations, including judgements. Since unnamed sequents refer to other rules in a set, they observe local judgements, allowing you to overload judgement forms in different sets.

For environment semantics, we create two sets, one to retrieve the value of a variable in the environment and one to update the value of a variable. Following the semantic specifications for variables enumerated above, we have the following set definitions:

```
set GETVALUE is

    judgement ENV |- VAR : INTEGER, ENV ;

    env[] |- variable V :
             integer 0, env[pair(variable V, integer 0)] ;

    env[pair(variable V, I) . env]
       |- variable V : I, env[pair(variable V, I) . env] ;

    env  |- variable V : n, env'
    ----------------------------
    env[pair(variable V', I) . env]
            |- variable V :
               n, env[pair(variable V', I) . env'] ;
      provided diff(V, V') ;

end GETVALUE ;
```

The first rule indicates that if we search for a variable in an empty environment, we create the variable with an initial value of 0. The value 0 and new environment are returned.

When the variable in the subject of the second rules matches the first variable in the context (the head of the list), the variable's value is returned along with the unchanged environment.

Otherwise, if the current variable is not at the head of the environment, the third rule recursively examines the rest of the environment and returns the recursively calculated value and environment. The premise (recursive call) is evaluated with only the tail end of the initial environment, named `env`.

Note the use of the TYPOL *provided* clause in the third rule. The provided part of a rule contains a list of conditions that must be met before a rule can be applied. In general, the expression:

```
provided premises
```

`premises` stands for a list of sequents separated by ampersands (`&`). The provided part of the third rule ensures that it will only be applied when the variable name of the subject differs from that at the head of the environment. **N.B.** We must add the function `diff` to our prolog file `expfunc.sp`. This function is defined as `expfunc$diff(A, B) :- A ~= B`. We must also **import** it in the TYPOL program.

In order to modify a variable's value in the environment, we define the `UPDATE` set.

```
set UPDATE is
    judgement ENV |- VAR, INTEGER : ENV ;

    env[] |- variable V, integer N :
             env[pair(variable V, integer N)] ;

    env[pair(variable V, _) . env]
       |- variable V, I : env[pair(variable V, I) . env] ;

    env  |- variable V, n : env'
    -------------
    env[pair(variable V', I) . env]
         |- variable V, n : env[pair(variable V', I) . env'] ;
       provided diff(V, V') ;
end UPDATE ;
```

The first rule adds a variable and the assigned value to an empty environment. The second rule means that if the first variable in the environment is the current variable, then the value of this variable is updated, whatever the old value (note use of underscore). In this case, the environment is otherwise unmodified. The last rule recursively searches the rest of the environment to find the current variable.

Having defined these two sets, we may complete the initial definition of `eval_exp` by adding the following rules for the `variable` and `assign` operators. Note the use of named sequents to explicitly refer to imported judgements.

```
env |- EXP : v, env' &
UPDATE(env' |- variable V, v : env'')
----------------
env |- assign(variable V, EXP) : v, env'' ;

GETVALUE(env |- variable V : v, env')
----------------
env |- variable V : v, env' ;
```

The first rule may be read:

> *To evaluate the operator* `assign`, *evaluate the expression in the current environment, update the variable with this value, and return the value and modified environment.*

while the second means:

> *To evaluate the variable* `V`, *find and return its value in the current environment and return the (possibly) modified environment.*

6.7.4 Exporting premises for the environment

With the introduction of the environment, we will want to export a second trigger proposition. The original trigger proposition allows us to evaluate a tree in an initially empty environment. However, we also want to export a proposition to allow us to evalute a tree in an existing environment. From scratch then, we can prove the first proposition given a tree and an empty initial environment (which spares us the effort of constructing an environment tree by hand). Then, we may use the resulting environment as the initial environment in a subsequent proof of the second proposition, and so on.

The first exported premise thus requires only the initial tree for the computation:

```
export env[] |- exp : I, env as init_eval_exp(exp) = env;
```

Note that the function `init_eval_exp` takes one argument, a tree, and returns the calculated environment.

We may go one step further. Rather than starting with an empty environment, we initialize it so that it contains the variable `RESULT` whose value is zero. After the computation, we update this variable with the result.[13] We modify the `export` clause accordingly:

```
export
   env[pair(variable "RESULT", integer 0)] |- exp : I, env  &
   UPDATE(env |- variable "RESULT", I : env')
  as init_eval_exp(exp) = env';
```

The second exported premise handles the non-empty environment:

```
export env |- exp : I, env' as full_eval_exp(exp, env) = env' ;
```

Note that in this case, the function `full_eval_exp` takes two arguments and returns the modified environment. This premise does not guarantee that the result of the evaluation will appear in the environment. For this, we explicitly assign the result to the variable `RESULT` as before:

```
export
   env |- exp : I, env' &
   UPDATE(env' |- variable "RESULT", I : env'')
  as full_eval_exp(exp, env) = env'';
```

N.B. We added an environment to some Typol rules without updating the judgement that allows Typol to type check them. We do so below.

[13] This insures that the `RESULT` variable will always be at the beginning of the list and therefore the first variable displayed, a purely aesthetic choice.

6.8 Sending error and warning messages from Typol

Often, TYPOL tools would like to emit errors and warning messages during a computation. CENTAUR offers a generic message mechanism that allows you to use resources to specify messages and display them in a special message view.

A *message module* is a resource file that contains any number of message specifications. Each entry of this file specifies a message by symbolic name. Each message has text which may contain arguments. When the message is posted from a TYPOL rule, a value must be provided for each argument. These values may be trees, strings, integers, etc. The first posted message causes a special message display window to open. Each message is displayed with its severity (Error, Warning, Info, etc.) followed by the text in which the formel arguments have been replaced by supplied values. When you post a message, it is also possible to assign a position in the source tree to the message or any of its arguments. These positions are highlighted in the source by a special selection. When you click on the message or argument in the message display, the current selection in the source moves to the associated position.

The TYPOL users manual provides more examples of posting messages. The message handling manual in the Core describes a wide range of message possibilities.

6.8.1 The `message` predicate

The `message` predicate posts a message from a TYPOL rule. This predicate must be imported from `message` as follows:

```
import message(string, string, string, string, path, string, _)
       from message ;
```

where the arguments are the following:

- the formalism name, here `"Exp"`.
- the message module name. We will call the message module for the interpreter,
  `interpreter`. It is possible to refer to any number of message modules in a program.
- the message name.

- the message class. Message classes may be any identifier, but the following identifiers are treated specially in the message display window: `Fatal`, `Error`, `Warning`, `Info`, and `Debug`.

- the message position. This is a path that designates a subtree in the source. Typically this is a descendent of the subject.

- supplementary information. In most cases, this information may be the empty string. However, if the information is the word `"private"` and the message has an associated location in the source, this location is not highlighted in the source. The location may be highlighted by clicking on the message in the message display.

- arguments. Each argument should be constructed by the syntactic construct `messarg`.

Since each argument has a value but may also have an associated position in the source and supplementary information, the `messarg` construction allows you to regroup this information when posting a message. This syntactic construction comes from a formalism named `messargs` and must be **use**d in a TYPOL program as any other formalism:

```
use messargs;
```

The `messarg` construct has the following syntax:

```
messarg(<value>, ref(<path>), info <string>)
```

where:

- The `<value>` is the message argument value. If the value is an abstract syntax tree stored in the TYPOL variable `TREE`, `<value>` must have the form `tree(TREE)`. If the value is a string, `<value>` must have the form `string <string-value>` (e.g., `string "7"`).

- The `<path>` is the position of the argument in the source tree. This is a path (e.g. `subject`). It may be `nil()` to mean that no position information should be associated with this argument. The value `nil()` is a syntactic construct from the **PSP** path language. If you use it in a TYPOL program, you must also **use** the PSP formalism at the beginning of the program.

- The last value is additional argument information, a string. Indicating `info "private"` makes the message argument private as explained above. Indicating `info ""` means that the message has no associated information.

6.8.2 Messages in Exp

Since the Exp interpreter is so small, there are not many places in the program where errors may occur. However, we would like to issue a warning message whenever a variable is added to the environment. Variables are added to the environment in the `GETVALUE` and `UPDATE` sets.

The message module file `contrib/Exp/messages/interpreter.rdb` was created when we generated the Exp hierarchy. The first thing we do is add a message specification to this file:

```
Centaur.Exp.MessageModules.interpreter.variable-added.Text: \
The variable {var} has been added to the environment.
```

This line specifies that for the **Exp** formalism, the message module **interpreter** contains the message **variable-added**. The message text contains one argument (indicated by {`var`}) which will be instantiated to the name of the variable added to the environment.

We modify the first axiom of the `GETVALUE` set to post a message when the variable is added to the environment:

```
env[] |- variable V :
         integer 0, env[pair(variable V, integer 0)] ;
   do message("Exp", "interpreter", "variable-added",
              "Warning", nil(), "",
             messarg(tree(variable V), ref(subject),
                     info "private")) ;
```

The value provided for the argument is the subtree `variable V`. The location of this Exp variable in the source is given by the subject of the axiom. We specify that the argument is private so that the location is not selected in the source until we click in the message display.

We use TYPOL's *do* construct to post the message. Any time the TYPOL motor proves a rule or axiom followed by a do clause, the do clause is executed.

We must similarly modify the first axiom of the UPDATE set:

```
env[] |- variable V, integer N :
         env[pair(variable V, integer N)] ;
    do message("Exp", "interpreter", "variable-added",
              "Warning", nil(), "",
             messarg(tree(variable V), ref(subject),
                     info "private")) ;
```

N.B. The PSP formalism defines a phylum named VAR, as does EXP. Using this formalism introduces an ambiguity into the GETVALUE and SETVALUE judgements, which refer to VAR. We must indicate that the VAR that interests us is the one from EXP, otherwise the TYPOL type checker will signal the error. The judgments thus become:

```
judgement ENV |- Exp::VAR : INTEGER, ENV;
```

6.9 The full Exp interpreter

Considering all this, here is the complete interpreter for EXP:

```
program eval_exp is
    use Exp;
    use PSP;
    use messargs;

    import add(integer, integer, integer),
           sub(integer, integer, integer),
           mult(integer, integer, integer),
           neg(integer, integer),
           diff(string, string) from expfunc ;

    import message(string,string,string,string,path,string,_)
           from message ;

    export
      env[pair(variable "RESULT", integer 0)] |- exp : I, env  &
```

```
  UPDATE(env |- variable "RESULT", I : env')
 as init_eval_exp(exp) = env';

export
  env |- exp : I, env' &
  UPDATE(env' |- variable "RESULT", I : env'')
 as full_eval_exp(exp, env) = env'';

judgement ENV |- Exp : INTEGER, ENV ;

env |- integer N : integer N, env ;

env |- EXP1 : integer x1, env'  &
env'|- EXP2 : integer x2, env''  &
add(x1, x2, x)
----------------
env |- plus(EXP1, EXP2) : integer x, env'' ;

env |- EXP1 : integer x1, env'  &
env'|- EXP2 : integer x2, env''  &
sub(x1, x2, x)
----------------
env |- minus(EXP1, EXP2) : integer x, env'' ;


env |- EXP1 : integer x1, env'  &
env'|- EXP2 : integer x2, env''  &
mult(x1, x2, x)
----------------
env |- prod(EXP1, EXP2) : integer x, env'' ;

env |- EXP : integer x1, env'  &
neg(x1, x)
----------------
env |- uminus(EXP) : integer x, env' ;

env |- EXP1 : v, env'
----------------
env |- exp_s[EXP1] : v, env' ;
```

```
  env |- EXP1 : v1, env'  &
  env'|- EXPS : v, env''
  ----------------
  env |- exp_s[EXP1.EXPS] : v, env'' ;

  env |- EXP : v, env' &
  UPDATE(env' |- variable V, v : env'')
  ----------------
  env |- assign(variable V, EXP) : v, env'' ;

  GETVALUE(env |- variable V : v, env')
  ----------------
  env |- variable V : v, env' ;

set UPDATE is

  judgement ENV |- Exp::VAR, INTEGER : ENV ;


  env[] |- variable V, I : env[pair(variable V, I)] ;
        do message("Exp", "interpreter", "variable-added",
                   "Warning", nil(), "",
                   messarg(tree(variable V), ref(subject),
                                 info "private")) ;

  env[pair(variable V, _) . env]
       |- variable V, I : env[pair(variable V, I) . env] ;

  env  |- variable V, n : env'
  -------------
  env[pair(variable V', I) . env]
       |- variable V, n : env[pair(variable V', I) . env'] ;
     provided diff(V, V') ;

end UPDATE ;

set GETVALUE is
```

```
    judgement ENV |- Exp::VAR : INTEGER, ENV ;


    message("Exp", "interpreter", "variable-added",
            "Warning", nil(), "",
            messarg(tree(variable V), ref(subject),
                    info "private"))
    ----------------
    env[] |- variable V : integer 0,
                          env[pair(variable V, integer 0)] ;

    env[pair(variable V, I) . env]
       |- variable V : I, env[pair(variable V, I) . env] ;

    env  |- variable V : n, env'
    -------------
    env[pair(variable V', I) . env]
         |- variable V : n, env[pair(variable V', I) . env'] ;
      provided diff(V, V') ;

  end GETVALUE ;
end eval_exp ;
```

6.9.1 Running the Exp interpreter

Once you have type checked, compiled, and written the above program, you may run it on a given source tree by calling the `ty-synchronous-run` function. This function takes a Le-Lisp function name (generated by an `export` clause) and initial values to be fed to the trigger proposition. The function returns a list whose elements may either be name/value pairs or message objects. The names are those exported as results. They appear in the Le-Lisp list of results in the same order.

N.B. TYPOL variables may contain single quotes, as in `env'`. The corresponding name in Le-Lisp may not contain single quotes, so each one is replaced by a dollar sign.

If `ty-synchronous-run` has not been loaded into your CENTAUR session, you may load it by calling:

```
(loadmodule 'centaur/tables/typol/run-time/batch/synch)
```

Thus, given a source VTP tree in the Le-Lisp variable `source-tree`, the following call:

```
(ty-synchronous-run '#:eval_exp:init_eval_exp source-tree)
```

would return a list of pairs. The call:

```
(setq env-tree                    ;;Final environment
   (cassq '|env$|
      (ty-synchronous-run '#:eval_exp:init_eval_exp
                          source-tree)))
```

assigns the value of the the TYPOL variable `env'` to the Le-Lisp variable `env-tree`. This value may then be used as the initial environment in the following call:

```
(setq env-tree                    ;;Final environment
  (cassq '|env$$|
   (ty-synchronous-run '#:eval_exp:full_eval_exp
                       source-tree
                       env-tree ;;Initial environment
                       )))
```

Section 7 discusses a graphic environment for calling the above TYPOL program.

7 Environment

To assist in the conceptual organization of a programming language environment, we introduce the notion of a *tool*. An environment is a set of communicating tools. Each tool performs one or more related tasks around a single source tree. Since the source tree is displayed in a ctedit, we often have the impression that it controls the environment. In fact, the source, though central to the environment, is passive. It is the tools —editors, pretty printers, type checkers, etc.— that initiate activity around the tree.

A tool is not a program. A tool is an object that performs a task by way of a program (in TYPOL, Le-Lisp, etc.). Each tool is an instance of a tool class. Tool classes specify the information that the tool keeps track of in order to function correctly.

For more information about tools and environments in general, please consult the environment guide in the Mantle.

7.1 Sophtalk

Tools should be as independent of each other as possible so that you can remove or modify them without disturbing the rest of the environment. The most important factors contributing to a healthy independence among tools are simple and complete functional and communication interfaces. To promote design that accounts for communication, CENTAUR makes extensive use of Sophtalk networks (see the Sophtalk documentation in the Core). Sophtalk is a system for enabling tools to communicate according to an *event model*. In this model, tools send signals to one another when important events occur. For example, signals may testify that a state change has occured (e.g., the editors tree has been modified), that an error has occurred, that data is required to complete a computation, etc.

Tools communicate within a network. Each tool is *encapsulated* within the network by an **stnode**. Each stnode belongs to an stnode class that determines the input signals the stnode receives and the output signals it may send when it belongs to a network. A tool may be encapsulated several times in the same network or in different networks. Tools may reside in the same process or run as different processes on remote machines.

Each signal may carry any number of values. A signal is received by all stnodes in the network with an input port having the same name.[14] The

[14]It is possible to rename ports when including an stnode in a network so that seman-

arrival of a signal at an stnode triggers an *input callback* function. This function's first two arguments are always the stnode itself and the name of the arriving signal. The callback must have same number of arguments after the stnode and signal name as the number of values carried by the signal. The callback may be defined to receive a list of values, however.

N.B. It is not always easy to determine what information should be memorized by an object and what information should be retrieved through network communication. For example, the source ctedit has no knowledge of the tools working on it. It should not be required to memorize these tools. It blissfully goes about its business while tools are being added or removed from the network. On the other hand, it is less obvious whether a tool working on the ctedit should memorize the ctedit in a structure or request information about the ctedit's root or selections through the network. In general, when an object requires information constantly from a known source, the object should memorize the source. If the source may be removed under the object's feet, or the source is unknown or many may reply, the tool should access information through network communication.

7.1.1 The variable network

The most important Sophtalk network used by CENTAUR is the *variable* network. This network has been constructed around the activities of the variable structure, namely tree and selection modification. Signals that circulate in this network include *modified*, which means the source tree has been modified, and *selection-modified*, which means that a selection has changed. To illustrate how these signals might be interesting to a listening network member, consider the following scenario. Each time the current selection changes, the clipboard reacts to the *selection-modified* signal and decides whether the operator of the subtree it contains matches the phylum of the currently selected subtree. The clipboard responds by announcing which editing operations are valid. The ctview listens to these signals and makes unavailable editing operations truly unavailable by disactivating the corresponding buttons in the **Edit** pulldown and turning the button titles gray.[15]

A variable usually participates in many variable networks, at least as many as there are ctedits that manipulate it. Thus, to recover the variable

tically compatible ports are connected.

[15] This unfortunately doesn't really happen.

network for a given ctedit, call the function `{ctedit}:network` (described in the environment manual).

7.1.2 The standard tool network

Another important network is called the *standard tool network* and is constructed around each tool in a language environment. The standard tool network contains encapsulations of the tool, the source, and a message handling mechanism. A few generic signals (*clear*, *terminate*), circulate within this basic network. We augment the standard network for a tool with additional encapsulations according to the tool's needs. For example, given a compiler tool, we might add an encapsulation of the tool that receives the signal *compile*, requesting the compiler to perform its task.

Figure 4 shows an approximation of a typical tool network. Stnodes are represented as ellipses. In this network, an encapsulated button sends the *compile* signal to the tool, who begins its calculation. Typically, a signal to start a computation is emitted by a graphical object (e.g., a button) that is encapsulated within the tool network. The graphical aspects of the button such as its location are independent of its communicative role.

Recall that the class of the encapsulating stnode determines the signals that the stnode receives and may send. And the same tool may be encapsulated any number of times by different classes of stnode.

While computing, the tool sends the predefined *post* signal that interests both the message display and source ctedit. This signal carries two values, a message object and a message identifier. The message display uses this data to display the message text. The source display uses this data to add message locations to a special message selection. The message display may emit the predefined *highlight-message* signal to tell the source ctedit to highlight a given message.

The source ctedit is encapsulated twice in the network. One encapsulation is predefined and concerns message needs. The other concerns functionalities offered by the tool. The tool may tell the source ctedit to highlight certain information (e.g., a variable name). This request might be transmitted by the user specified signal *highlight-data*. When we introduce a new functionality into a network, we tend to re-encapsulate members rather than modifying stnode classes to allow for new signals. This way, functionalities may be added, removed, or modified, without affecting one another.

The environment manual of the Mantle documents the functions that create the standard tool network and allow you to add buttons to it easily.

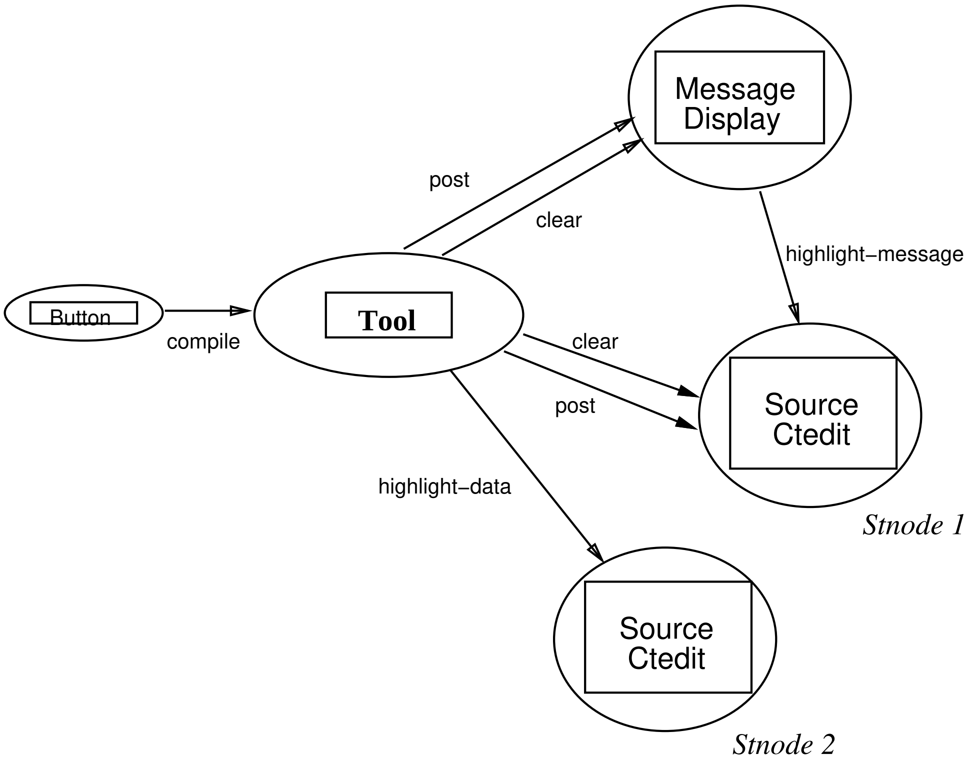

Figure 4: A typical tool network.

7.1.3 Participating in a network

In order to construct a language environment, you will almost always be required to participate in network communication. To do so you must:

- Find the network that interests you, and figure out which signals circulate within this network and for what events.
- Define an stnode class whose input and output ports correspond to the signals you want to listen to or emit within the network. This is done by the function `st-declare`.
- Create an instance of this class with the function `st-create`.
- Encapsulate the object you wish with the stnode instance. You may use the function `st-store-and-set-action` to simultaneously encapsulate the object and define the input callback package for the stnode. When an input signal arrives on the stnode, the name of the input callback function called is the callback package followed by the port name. Once an stnode encapsulates an object, you may retrieve the object by calling `(st-object stnode)` (usually done in the callback function whose first argument is the stnode).
- Include the stnode in the network with the function `st-include`.

Below we demonstrate this procedure.

7.2 The environment module

The Le-Lisp code for the Exp environment should be stored in the directory `contrib/Exp/environment` (specified by resources in the formalism resource file). This directory contains a module description file named `module.LM` which allows us to compile the Le-Lisp code and which directs the loading of the Exp environment. The code will be stored in two other files:

- `Exp.ll`: This code sets up general aspects of the the Exp environment.
- `interpreter.ll`: This code builds the environment concerning the Exp interpreter.

We will not discuss the compilation of the Exp module in this tutorial. However, in order for Centaur to automatically load the environment module, make sure that the `module.LM` file contains the following text:

```
defmodule contrib/Exp/environment

files ("contrib/Exp/environment/Exp.ll"
       "contrib/Exp/environment/interpreter.ll"
       )
import (centaur/tables/typol/env)
```

Note that we import the TYPOL environment module. This ensures that CENTAUR loads TYPOL environment code necessary for the construction of the EXP interpreter.[16]

We do not provide a listing of the final version of the EXP environment module in this document. The CENTAUR distribution includes the EXP hierarchy, so you may consult the complete module specification in the directory `contrib/Exp/environment` below the CENTAUR root directory.

7.3 Environment setup and cleanup

When you read a program in a language `L` into a new ctview or one containing a program in a different language, CENTAUR calls the function:

```
({L}:set-environment <ctview>)
```

if it exists, to install the `L` environment. Installation involves the creation of tools, tool networks, graphical changes to the ctview, etc.

Similarly, before changing environments in a ctview or killing a ctview, CENTAUR calls the function:

```
({L}:clear-environment <ctview>)
```

so that you may clean up the installed environment.

We define the following functions in `contrib/Exp/environment/Exp.ll`

The first step towards building the EXP environment is to define the `{Exp}:set-environment` function. This function should be very simple — just general initialization and then calls to functions that construct each specified tool. Our setup function would thus be:

[16] The environment may have already been loaded if we have, say, edited a TYPOL program. It is not reloaded in this case.

```
;;Environment for Exp.
(setq #:sys-package:colon '#:Exp-environment)

;;;;;;;;;;;;;;;;
;;Setup
;;;;;;;;;;;;;;;;
(de {Exp}:set-environment (ctview)
    (:initialize-environment ctview)
    (#:Exp:interpreter:create ctview)
    ctview
    )
```

We will define the function that creates the interpreter shortly. To initialize the ctedit and ctview that contain the Exp program, we:

- Name the ctedit. This name may be used in resource specifications so that we can write other resource specifications that only apply to those editors containing Exp programs.
- Create a popup menu. The popup menu is associated with the ctedit. Later we will specify a mouse binding through resources that makes the popup appear in any editor containing an Exp program. The popup menu is stored in a global variable during the construction of the environment so that any tool (here, only the interpreter) may add buttons to it.

```
;;;;;;;;;;;;;;;;
;;Initialization
;;;;;;;;;;;;;;;;
(defvar :popup ())

(de :initialize-environment (ctview)
    (let ((ctedit (send 'object ctview)))
       ;;Name the ctedit for resources.
       (#:properties:set-user-property ctedit 'Editor 'Exp)
       ;;Create a popup and assign it to the current ctedit.
       (setq :popup ({menu}:create 0 0 'column))
       ({ctedit}:popup ctedit :popup)
       ;;Redraw to take new resources into account.
       ({ctedit}:redraw ctedit)
       ))

;;Convenience function used by tools.
(de :handle-button (network title sig)
  (let ((but (create-std-environment-button network title () sig t)))
```

```
    ({menu}:add-button :popup but)
    ))
```

The `create-std-environment-button` function creates a button and adds it to a tool network. When activated, the button emits the specified signal, which is received by an stnode encapsulating the tool. This function only creates the button and inserts it in the network. Our convenience function then adds the button to the popup menu.

Finally, since a ctedit may be reused for programs belonging to a different formalism, we must define a cleanup function to remove the ctedit name, remove the popup menu, and kill all standard tool networks.

```
;;;;;;;;;;;;;;;;
;;Cleanup
;;;;;;;;;;;;;;;;

(de {Exp}:clear-environment (ctview)
    (let ((ctedit (send 'object ctview)))
      ;;Clear the ctedit name.
      (#:properties:reset-properties ctedit)
      ;;Clear the popup
      ({ctedit}:popup ctedit ())
      ;;Kill all tool networks.
      (cleanup-std-environment ctview)
      ))
```

7.4 The interpreter tool network

The Exp environment will consist of one tool, an interpreter. First we consider the tool structure the network in which it communicates. The functions described below will be stored in the file:

`contrib/Exp/environment/interpreter.ll`

7.4.1 The standard Typol tool

Since the interpreter tool must communicate with Typol, it should be an instance of the class of tools designed for such communication (or a subclass). The standard Typol tool class defines the following properties (among others):

- *name*: The tool name, here `interpreter`.

- *formalism-name*: The name of the formalism around which the tool operates, here `Exp`.
- *source*: The source ctedit.

7.4.2 The interpreter tool

In the Exp environment, we must subtype the standard Typol tool class to allow the interpreter tool to memorize information required for its normal operation. In addition to the source tree (available through the source ctedit), the tool must keep track of an initial environment tree. Rather than store just the tree, we prefer to store a Centaur variable structure because later we will want to modify the input Vtp tree and this should be performed through the Centaur variable interface.[17]

Tool classes are defined as Le-Lisp structures, so we define a subtype of the standard Typol class with the following code:

```
;;;;;;;;;;;;;;
;;Tool creation
;;;;;;;;;;;;;;
(setq #:sys-package:colon '#:Exp:interpreter)

(eval-when (load eval compile)
   (defstruct #:centaur-tool:formalism:typol:std:Exp
              initial-environment))

;;We could also pass an initial environment tree to the
;;following function.

(de :tool:create (ctedit)
  (let ((new (#:centaur-tool:formalism:typol:std:Exp:make))
        (variable ({variable}:make ()))
        )
     ;;Create the variable network [not done by
     ;;the {variable}:make function].
     ({#:stnode:variable}:create variable)
     ;;Initialize tool.
     (send 'name new 'interpreter)
     (send 'formalism-name new 'Exp)
     (send 'source new ctedit)
     (#:centaur-tool:formalism:typol:std:Exp:initial-environment
          new variable)
```

[17]Among other reasons, this gives us free redisplay in any ctedit's that display this variable.

```
     new))

(de :tool:get-source (tool)
    ({variable}:root ({ctedit}:subject (send 'source tool))))

(de :tool:get-initial-env (tool)
    ({variable}:root (send 'initial-environment tool)))

(de :tool:set-initial-env (tool env-tree)
    (let ((variable (send 'initial-environment tool)))
      ;;This call provokes a redisplay in any ctedits
      ;;containing this variable.
      ({variable}:change variable
                         ({variable}:root variable)
                         env-tree)))
```

We have also defined a convenient functions to create an initial tool, get the source tree, and get and set the tree used as the initial environment.

7.4.3 The tool network

The function that creates the interpreter tool network must create an instance of the tool class defined above. It must include the tool in a standard Typol tool network then add other network components specific to the Exp environment to send and receive signals.

In order to include the tool in the standard network, we need to define an stnode class, then create an instance to encapsulate the tool. We define the following stnode class so that evaluation may be triggered by the signal *interpret*:

```
(st-declare Exp-interpreter
            ;;Input signals
            (interpret)
            ;;Output signals
            ())
```

Now we may create the interpreter tool network:

```
;;;;;;;;;;;;;;
;;Tool network creation
;;;;;;;;;;;;;;
(setq #:sys-package:colon '#:Exp:interpreter)

(de :create (ctview)
```

```
    (lets ((ctedit (send 'object ctview))
           (tool (:tool:create ctedit))
           (network (ty-tool-create-std-network tool
                                                 'Exp-interpreter
                                                 ':callback))
           )
         ...))
```

The specified callback package `':callback` determines what functions are called when an input signal arrives. So far, the only callback signal is *interpret*, so the callback function invoked will be `':callback:interpret`.
Now that we have created the standard tool network and the interpreter is capable of receiving the *interpret* signal, we need to add an encapsulated button to the network capable of emitting that signal to start evaluation. In the following, we add a button whose title is `Eval Exp` and who emits the signal *interpret* when activated:

```
(de :create (ctview)
    (lets ((ctedit (send 'object ctview))
           (tool (:tool:create ctedit))
           (network (ty-tool-create-std-network tool
                                                 'Exp-interpreter
                                                 ':callback))
           )
         (:create-decoration ctview network)
         ))

(de :create-decoration (ctview network)
    ;;Use the convenience function of Exp.ll
    (#:Exp-environment:handle-button network "Eval Exp" 'interpret))
```

We still must define the callback function triggered by the *interpret* signal. In this function, the tool must call the Typol program with the correct initial values: a source tree and an optional environmen tree. A tool calls a Typol program with the the generic function `ty-run`. This function's arguments are a tool, a program name (as generated by the `export` predicate), a debug flag, and initial values.

```
;;Later we will set this value by resources.
;;When true, this variable causes the interpreter
;;to be executed in debug mode.
(defvar :debug ())

(de :callback:interpret (stnode port)
```

```
    (lets ((tool (st-object stnode))
           (root (:tool:get-source tool))
           (env  (:tool:get-initial-env tool))
           )
      ;;If no environment tree memorized by tool,
      ;;call appropriate Typol predicate.
      (if env
          (ty-run tool '#:eval_exp:full_eval_exp :debug root env)
          (ty-run tool '#:eval_exp:init_eval_exp :debug root)
          )))
```

7.4.4 Displaying results simply

Let's sum up where we are: We can now trigger evaluation —with or without an initial environment— from a button in the Exp popup menu. A message display window will open automatically for all warning messages sent by the interpreter. However, we have done nothing to display either the initial environment or the results. In a simple version of the environment, we merely want to display the result environment computed by Typol. To do so, we make use of the function `ty-result-network`. This function adds stnodes to a Typol tool network so that a view opens automatically for a specified result.

```
(de :create (ctview)
    (lets ((ctedit (send 'object ctview))
           (tool (:tool:create ctedit))
           (network (ty-tool-create-std-network tool
                                                'Exp-interpreter
                                                ':callback))
           )
       ;;Handle results
       (:create-result-view-spy tool network)
       (:create-decoration ctview network)
       ))

(de :create-result-view-spy (tool network)
    (ty-result-network network tool '|env$| "Result view env'")
    (ty-result-network network tool '|env$$| "Result view env''")
    )
```

The third argument of `ty-result-network` is the name of the result to listen for, as named in the exported Typol clause. Since we export two clauses, we plan for two different views to open. These views reopen automatically if killed.

7.5 A complete Exp environment

The previous code suffices for a simple Exp environment. Now we present the specifications for a more complete Exp environment. We have tried to make the environment as modular as possible so that users may borrow ideas for their own environments according to the complexity they desire.

The environment will still contain only one tool, an interpreter, with which we will be able to:

- Evaluate an Exp program in an initial environment that may be empty.
- Trigger evaluation through a popup menu. **N.B.** During the the evaluation, the popup menu is replaced by another with an *Abort* button to interrupt the Typol engine.
- Display the initial environment. This graphic interface allows us to edit the initial environment.
- Display the result environment. This view will open automatically after the first evaluation, then reopen if killed.
- Clear the initial and result environments (including the views that display them) from the popup menu.
- Display warning messages from the Typol program in a message display. This service is offered for free by the standard tool network.
- Select a variable name in the source ctedit or either environment display and highlight all occurrences of this variable name in all windows.
- Kill the source ctedit and automatically kill the environment views.

N.B. To visualize the source select that highlights interpreter warnings, you must add the following select resource specifications to the file `contrib/Exp/syntax/Exp.rdb`:

```
?.?.visual-color.?.formalism-Exp.?.Format.select-interpreter-messages.background: \
PaleGoldenrod
?.?.visual-mono.?.formalism-Exp.?.Format.select-interpreter-messages.mode: none
```

Remember to reset the resources.

To realize the above specifications, we may use the same tool structure defined above for the complete environment. The main difference between the simple and complex environment stems from modifications of the tool network.

7.6 Views

So far, we have no way to easily create or display an initial environment, and no way to display results (supposing that we undo the simple result display). We will accomplish these tasks by adding stnodes to the tool network. We modify the interpreter tool creation function to include appropriate encapsulations in the tool network:

```
(de :create (ctview)
    (lets ((ctedit (send 'object ctview))
           (tool (:tool:create ctedit))
           (network (ty-tool-create-std-network tool
                                                'Exp-interpreter
                                                ':callback))
           )
       ;;Add Exp specific stnodes.
       (:create-initial-view-spy tool network)
       (:create-result-view-spy tool network)
       ;;Graphic interface.
       (:create-decoration ctview network)
       ))
```

There is a subtle point which we must consider. The trees that will be displayed by the initial and result environment views belong to the Exp formalism. If we are not careful, when we open a view on an environment tree, the Exp environment will be applied to the view (e.g., the popup menu will be available) as for any Exp source program. Thus, environment views must be opened without loading the standard Exp environment.

7.6.1 Manual initial view creation

Our specifications state that we must be able to open a view on the tool's initial environment through the popup menu. We must also be able to clear the initial environment through a button in the popup menu. We will do so by adding elements to the tool network to emit and receive the appropriate signals.

First, we add the button to the decoration creation function to emit the signals *open-view* and *clear*:

```
(de :create-decoration (ctview network)
    ;;Use the convenience function of Exp.ll
    (#:Exp-environment:handle-button network "Eval Exp" 'interpret)
    (#:Exp-environment:handle-button network "Init View" 'open-view)
    (#:Exp-environment:handle-button network "Clear" 'clear-views))
```

Then, in the function `:create-initial-view-spy`, we encapsulate the tool to listen for these signals.

```
;;;;;;;;;;;;;;;;
;;Initial view
;;;;;;;;;;;;;;;;
(st-declare Exp-initial-env (open-view clear-views) ())

(de :create-initial-view-spy (tool network)
    (let ((stnode (st-create 'Exp-initial-env)))
       ;;Encapsulate tool.
      (st-store-and-set-action stnode tool ':initial-env-callback)
       ;;Include both stnodes in tool network.
      (st-include network stnode)
      ))

(de :initial-env-callback:open-view (stnode port)
    (lets ((tool (st-object stnode))
           (tool-network (st-up stnode))
           (var (send 'initial-environment tool))
           (ctview (with ((current-manager
                            #:gfxobj:look:ctview:plain-manager))
                       (map-ctview-var "Initial view" var
                                       () () () () () ())))
           (view-network (:env-view:adjust ctview tool-network))
           )
       ;;Remove "File" pulldown from ctview (position 0).
       ({editor-view}:delete-menu ctview () 0)
       ;;Make view network depend on principal network
       ;;so that initial env view dies when source dies.
       (st-dependent-network tool-network view-network)
       view-network
       ))

(de :initial-env-callback:clear-views (stnode port)
```

```
(lets ((tool (st-object stnode)))
   ;;Clear initial environment tree and redisplay ctedits.
   (:tool:set-initial-env tool ())
   ;;Clear old messages and ctedit source selections.
   (clear-std-environment tool)
   ))
```

N.B.

- Recall that a CENTAUR variable structure may be displayed by several views. We open a normal ctview on a CENTAUR variable structure with the function `map-ctview-var`. The first argument is the view title, the second a variable structure, and the others may be `()`. Note that with this architecture, it is possible to open several views on the initial environment.

- Opening the view with the *current-manager* positionned as indicated prevents CENTAUR from loading the EXP environment.

- Given an stnode, the Sophtalk function `st-up` returns the network in which the stnode communicates. In this case, the network is the tool network, to which we will add other stnodes shortly.

- The `st-dependent-network` function makes a "secondary" network dependent on a "primary" network; when the primary network dies (here, the tool network), the dependent network is terminated (thus closing the initial environment view).

- The `clear-std-environment` is a generic function that empties message views and the message related selections in source views.

- Since we want to furnish editing functionalities to the user (both structural and in-line) via this view, but no read or write options, we remove the $\boxed{\textit{File}}$ pulldown from the menu bar.

7.6.2 Automatic result view creation

We have specified that we wish to open a view by necessity to display the result environment. This problem relates to Sophtalk since results from TYPOL are transmitted by the *result* signal. We must therefore insert an stnode in the network to listen for this signal and open a result view. However, we only want to open one view, and this view should be updated by listening to subsequent *result* signals.

The *result* signal carries two values: the name of the result and its value. The name of the result is the variable name as it is returned by an exported clause. When an exported premise returns several results, each is sent to the tool network by a separate *result* signal. In our case, we have only one result (whose Le-Lisp name is `'|env$|` for an evaluation without initial environment and `'|env$$|` otherwise): an environment tree.

To open a window when a signal arrives, we make use of the function `st-automatic-create`. This function creates a spy in a principal network that listens to a given signal. When the signal arrives, the spy performs some action such as opening a new view. The spy sees to it that the action is performed only once, even if the same signal arrives later. The action must return a Sophtalk network, which then "depends" on the principal network. When the primary network dies (here, the tool network), the dependent network also dies automatically (here, a result view).

The function `st-automatic-create` takes a signal name to listen for, a callback function name, and initial data. When the signal arrives, the callback function is called with the following arguments: the signal name, the intial data, and any data carried by the signal. The initial data is any data known when `st-automatic-create` was called but which will not be available when the trigger signal arrives.

Here is the code necessary to open the result environment view automatically when the first *result* signal arrives:

```
;;;;;;;;;;;;;;;;
;;Result view
;;;;;;;;;;;;;;;;
(de :create-result-view-spy (tool network)
  ;;The tool and network are known at creation time,
  ;;but not easily available when the run signal arrives!
  ;;Thus, we pass them on as initial data.
  (st-automatic-create network 'result ':result-view-callback
                       tool network))

(de :result-view-callback (port tool network name env-tree)
  (lets ((view (with
              ((current-manager #:gfxobj:look:ctview:plain-manager))
                 (map-ctview "Result view" env-tree
                             () () () () () ())))
         (ctedit (send 'object view))
         (view-network ({ctedit}:network ctedit))
```

```
        )
       ;;This function MUST return a network.
       view-network
      ))
```

The function `map-ctview` opens a view given a VTP tree and view title.

According to the semantics of `st-automatic-create`, the second function is only called once to open the view. Subsequent *result* signals are not handled in any special way, so the result view will not be updated after later evaluations. To accomplish this, we must add an stnode to the tool network that listens to the second and all succeeding *result* signals. Below we define a new stnode class `Exp-result-spy`, create an instance which encapsulates the result ctedit, and include it in the tool network.

```
(st-declare Exp-result-spy (result) ())

(de :result-view-callback (port tool network name env-tree)
  (lets ((view (with
               ((current-manager #:gfxobj:look:ctview:plain-manager))
                   (map-ctview "Result view" env-tree
                               () () () () () ()))))
         (ctedit (send 'object view))
         (view-network ({ctedit}:network ctedit))
         (stnode (st-create 'Exp-result-spy))
         )
       ;;Create stnode to listen to 'result signal
       (st-store-and-set-action stnode ctedit
                                ':result-view-callback)
       ;;Include in tool network.
       (st-include network stnode)
       ;;Make stnode dependent on view network
       (st-dependent-stnode view-network stnode)
       ;;This function MUST return a network.
       view-network
       ))

(de :result-view-callback:result (stnode port name env-tree)
    (lets ((ctedit (st-object stnode))
           (var ({ctedit}:subject ctedit))
           )
        ;;This call redisplays result view.
        ({variable}:change var ({variable}:root var) env-tree)
        ))
```

Note the call to `st-dependent-stnode`. The result stnode listens for signals in the tool network, but depends on the life of the view network. If the view is closed, the result stnode must be removed from the primary network. Otherwise the result stnode would continue to receive the *result* signal and try to update a view that no longer exists. Using **st-dependend-stnode** ensures that the result stnode will be removed from the primary network should the secondary network die.

We still lack the ability to clear the result view by signal. We have already added a button to the popup to emit the *clear-views* signal, so we need only encapsulate the result view to heed it. Since we have already encapsulated the result view in the tool network, we will simply add another input signal to the stnode class and define an appropriate callback function.

```
(st-declare Exp-result-spy (result clear-views) ())

(de :result-view-callback:clear-views (stnode port)
    (lets ((ctedit (st-object stnode))
           (var ({ctedit}:subject ctedit))
           )
      ;;Redisplay automatic.
      ({variable}:change var ({variable}:root var) ())
      ))
```

Figure 5 shows the stnodes that we have added for to the standard tool network for the interpreter. This diagram does not show the message display, which communicates with the tool in a separate network.

7.7 Highlighting data

Almost all of our environment specifications have been satisfied. The last task is to be able to select a variable in any of the interpreter's windows and see all occurrences in every window. To do so, we must:

- Re-encapsulate each view in the tool network to send and receive highlight requests.
- Add a selection to each view to memorize all occurrences of a variable name. We also add a select to each ctedit to display this selection information.
- Write the selection algorithm that calculates the selection path in a source tree for a given variable name.
- Attach an action to the mouse button to trigger the selection calculation when a variable has been selected.

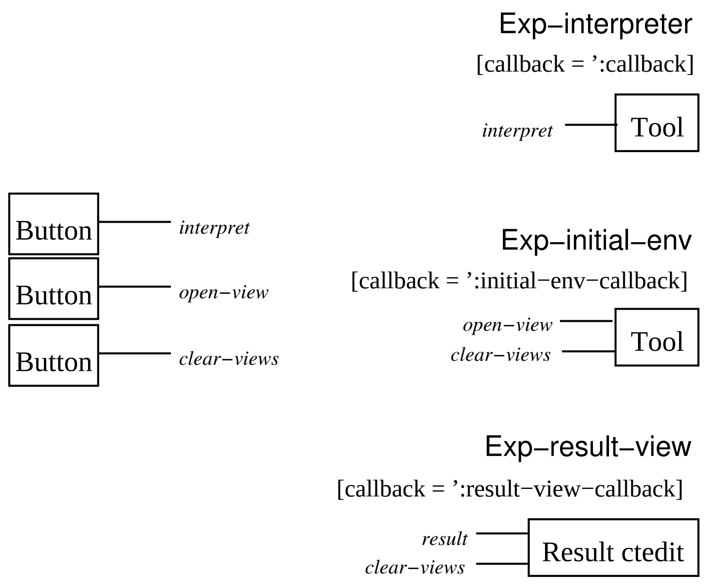

Figure 5: The EXP interpreter tool network contains encapsulations of buttons, two different encapsulations of the interpreter tool (one to start evaluation and one that deals with the initial environment view), and an encapsulation of the result view.

- Specify select resources for the highlighted data.

7.7.1 View communication

According to our specifications, each view must be able to tell the other two to highlight all occurrences of a given variable name. The scenario is completely symmetric: the stnode that encapsulates each view will be the same since each view should be able to emit and receive the *highlight-variable* signal. However, in Sophtalk, an stnode may not have an input and output port bearing the same name. To implement the kind of symmetry we require, we must introduce a neutral stnode that listens to a highlight request signal, then broadcasts a signal with a different name, received by the three view stnodes. The following stnode classes accomplish this:

```
(st-declare Exp-highlight-request-stnode (hi-var-request) (hi-variable))
(st-declare Exp-highlight-stnode (hi-variable) (hi-var-request))
```

An instance of the first class will encapsulate a neutral highlight request manager. An instance of the second class will encapsulate each of the views (source and environment). Each instance of the second class will encapsulate a CENTAUR variable structure so that a single callback function may be used for all source and environment views.

```
;;;;;;;;;;;;;;
;;Variable highlighting
;;;;;;;;;;;;;;
(setq #:sys-package:colon '#:Exp:interpreter)

(st-declare Exp-highlight-request-stnode (hi-var-request) (hi-variable))
(st-declare Exp-highlight-stnode (hi-variable) (hi-var-request))

;;;;;;;;;;;;;
;;In source.
;;;;;;;;;;;;;
(de :create (ctview)
    ...
       ;;Add new stnodes for highlighting in the source view.
       (:create-tool-highlight-spy tool network)
       ...
       ))

;;;
;;;Define request manager structure.
;;;
```

```
(eval-when (load eval compile)
     (defstruct :manager))

(de :create-tool-highlight-spy (tool network)
  (let ((req-stnode (st-create 'Exp-highlight-request-stnode))
        (hi-stnode  (st-create 'Exp-highlight-stnode))
        (manager (:manager:make))
        (ctedit (send 'source tool))
        (var ({ctedit}:subject ctedit))
        )
    ;;Encapsulate source variable to listen for highlight.
    (st-store-and-set-action hi-stnode var ':hi-callback)
    ;;Encapsulate manager to receive requests.
    (st-store-and-set-action req-stnode manager ':hi-manager-callback)
    ;;Include both stnodes in tool network.
    (st-include* network hi-stnode req-stnode)
    ))
```

Then we encapsulate the variable of each type of environment view:

```
;;;;;;;;;;;;;
;;In initial view.
;;;;;;;;;;;;;

(de :initial-env-callback:open-view (stnode port)
    (lets ((tool (st-object stnode))
           (tool-network (st-up stnode))
           (var (send 'initial-environment tool))
           (ctview (with ((current-manager
                            #:gfxobj:look:ctview:plain-manager))
                      (map-ctview-var "Initial view" var
                                       () () () () () ())))
           (ctedit (send 'object ctview))
           (view-network ({ctedit}:network ctedit))
           (hi-stnode (st-create 'Exp-highlight-stnode))
           )
      ...
      ;;Since Exp environment not loaded for this view,
      ;;we must name the ctedit for mouse event resources.
      (#:properties:set-user-property ctedit 'Editor 'Exp)
      ;;Encapsulate initial view var to listen for highlight.
      (st-store-and-set-action hi-stnode var ':hi-callback)
      (st-include (st-up stnode) hi-stnode)
      ;;Make hi-stnode dependent on view-network
      (st-dependent-stnode view-network hi-stnode)
      view-network
      ))
```

```
;;;;;;;;;;;;;
;;In result view.
;;;;;;;;;;;;;
(de :result-view-callback (port tool network name env-tree)
  (lets ((view (with
                 ((current-manager #:gfxobj:look:ctview:plain-manager))
                     (map-ctview "Result view" env-tree
                                 () () () () () ())))
         (ctedit (send 'object view))
         (var ({ctedit}:subject ctedit))
         (view-network ({ctedit}:network ctedit))
         (stnode (st-create 'Exp-result-spy))
         (hi-stnode (st-create 'Exp-highlight-stnode))
         )
        ...
        ;;Since Exp environment not loaded for this view,
        ;;we must name the ctedit for mouse event resources.
        (#:properties:set-user-property ctedit 'Editor 'Exp)
        ;;Encapsulate result view var to listen for highlight.
        (st-store-and-set-action hi-stnode var ':hi-callback)
        ;;Include in tool network.
        (st-include* network stnode hi-stnode)
        ;;Make stnodes dependent on view network
        (st-dependent-stnode view-network stnode)
        (st-dependent-stnode view-network hi-stnode)
        ;;Must return a network.
        view-network
        ))
```

Note that in addition to adding the highlight stnodes, we have specified the *Editor* property for each view ctedit. This was done in {`Exp`}`:set-environment`, but this function is not called for environment views. By naming these views `Exp`, we may share the same mouse event list as specified for ordinary source views.

Finally, we define the highlight callback functions:

```
;;;;;;;;;;;;;;;;;;;;;;;;;
;;Callbacks
;;;;;;;;;;;;;;;;;;;;;;;;;
;;The manager callback function
(de :hi-manager-callback:hi-var-request (stnode port variable-name)
    (st-emit stnode 'hi-variable variable-name))


;;The common highlight callback function. Supposes encapsulated
;;object is a variable.
```

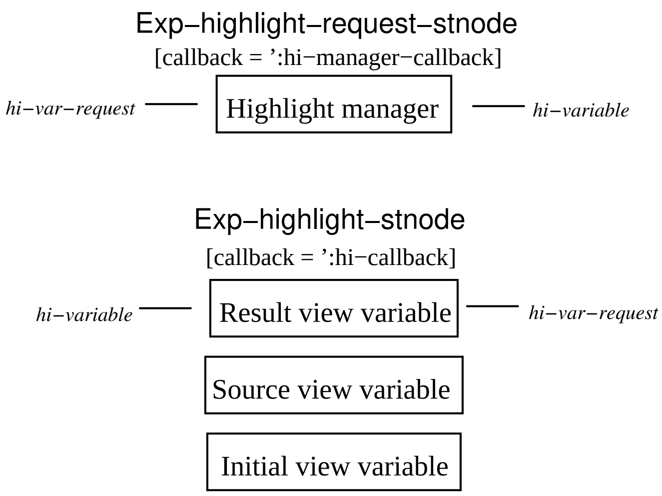

Figure 6: Encapsulations for highlighting variables in all views.

```
(de :hi-callback:hi-variable (stnode port variable-name)
    (:select-variables (st-object stnode) variable-name))
```

Figure 6 summarizes the different encapsulations made for highlighting. We define `:select-variables` below.

7.7.2 Adding a selection

We must now add a highlight selection to the variable structures containing the source tree and the two environment trees. We name this selection `{current}:highlight-variable` (see the path manual for information about selections). In order to be visible, the selection must be displayed by a select in the ctedit. We choose the select type `{textual}:all` to display our variable names (see the ctedit manual for information about selects).

The interpreter tool must add the selection to the source ctedit:

```
(de :create (ctview)
    (lets ((ctedit (send 'object ctview)))
       ...
       ;;Create the highlight select and selection.
       ({ctedit}:create-select ctedit
```

```
                                '{current}:highlight-variable
                                '{textual}:all)
         ...
         ))
```

We do the same thing in each function that creates an environment view:

```
(de :initial-env-callback:open-view (stnode port)
    (lets (...
           (ctedit (send 'object view))
           ...
           )
      ;;Create the highlight select and selection.
      ({ctedit}:create-select ctedit
                              '{current}:highlight-variable
                              '{textual}:all)
      ...
      ))
```

The same code must be added to `:result-view-callback`.

7.7.3 The selection algorithm

Next we must consider the selection algorithm. When we click with the mouse, how do we recover the variable name? And given a variable name, how do we traverse the source tree and memorize all occurrences in a selection? The selection algorithm will apply equally to the source and environment views.

First, let's suppose we have a variable name. To calculate occurrences of this name in an Exp tree, we traverse the Exp tree via the function `{tree}:select-all-true` described in the path manual:

```
(de :calculate-variable-path (tree varname)
    (when varname
       ({tree}:select-all-true
          tree
          (lambda (subtree)
              (when (eq 'variable ({tree}:operator-name subtree))
                    (eqstring varname ({tree}:atom_value subtree))
                    )))))
```

Every time the `lambda` function returns true, the function adds a subtree to the selection path. We simply test whether the subtree's associated operator is a **variable**. When the variable name matches the argument variable name, the function returns true.

The result of this function is a *path*. The path must be stored in the highlight selection. When you change the value of a selection, you must explicitly announce the change by calling {variable}:selection-modified. One effect of this function is to redisplay the select in every ctedit that displays the selection. Thus, let us encapsulate the previous function so that we have the CENTAUR variable structure available:

```
(de :select-variables (var varname)
    (let ((root ({variable}:root var)))
      ({selection}:change
        ({variable}:find-selection var '{current}:highlight-variable)
        (:calculate-variable-path root varname))
      ({variable}:selection-modified var '{current}:highlight-variable)
      ))
```

7.7.4 Triggering an action with the mouse

The only remaining issues are how to trigger a function through the mouse and how to recover a variable name from a selected subtree.

The mouse event list specifies a callback package for a given mouse button. Le-Lisp calls the functions whose names are created by concatenating the callback prefix to `mouse-down`, `mouse-drag`, and `mouse-up` when these events occur. In section 3.1.1, we bound the callback package {Exp}:select-variable to the middle mouse button. Therefore, we must define the following functions:

```
(de {Exp}:select-variable:mouse-down (ctedit event))
(de {Exp}:select-variable:mouse-drag (ctedit event))
(de {Exp}:select-variable:mouse-up (ctedit event))
```

When we click with the mouse, the ctedit, using the position information provided by the Le-Lisp event structure, calculates a path from the root of the graphic structure to the selected token or tokens. The call:

```
(send 'image-path <ctedit> <event>)
```

returns the path in the graphic structure for a given Le-Lisp event, called the *image path*. We may then convert this path into one that identifies the corresponding position in the abstract syntax tree.[18] The call:

[18] The correspondence depends on the type of formatter being used.

```
({ctedit}:get-subject-path <ctedit> <image-path>)
```

returns the structural path for the given image path. This path points to a subtree, so we need only retrieve the subtree, and if its associated operator is `variable`, emit the signal *hi-var-request* in the tool network. This is done in the mouse-down function:

```
(de {Exp}:select-variable:mouse-down (ctedit event)
    (lets ((image-path (send 'image-path ctedit event))
           (subject-path ({ctedit}:get-subject-path ctedit image-path))
           (subtree ({path}:first-value subject-path))
           )
       (stobject-emit
         ({ctedit}:subject ctedit)
         '#:Exp-highlight-stnode:hi-var-request
         (when subtree
            (let ((operator-name ({tree}:operator-name subtree)))
               (when (eq 'variable operator-name)
                 ({tree}:atom_value subtree)))))
       ))
```

Note that we emit the signal from the ctedit, and not directly from the stnode that encapsulates it with the function `stobject-emit`. This function's first argument is the encapsulated object. The second argument is full signal name composed of the the class of the stnode used to encapsulate the object and the desired signal name. We emit the *hi-var-request* signal even when the mouse doesn't select a subtree or the selected subtree does not have the associated operator `variable`. This allows us to clear the highlight selection with the mouse.

We have not specified mouse-drag or mouse-up behavior here. However, you might want to change the mouse behavior so that the variable is selected only when the mouse button is released, not depressed. Or you might want to change the selection as long as the mouse is dragged, etc.

7.7.5 Select resources

Since several selects now appear in each of our ctedits —for the current selection, the interpreter warning message selection, and the variable highlight selection— we must fine tune their presentation so that we may distinguish them. First we add a few resource specifications to the file:

`contrib/Exp/pprinters/pprinters.rdb`

```
*visual-mono*formalism-Exp.Pprinter.Format.select-highlight-variable.Mode: t
*visual-color*formalism-Exp.Pprinter.Format.select-highlight-variable.foreground:\
PaleGoldenrod
*visual-color*formalism-Exp.Pprinter.Format.select-highlight-variable.background:\
ForestGreen
*formalism-Exp.Pprinter.Format.select-highlight-variable.Mode: static
*formalism-Exp.Pprinter.Format.select-highlight-variable.Priority: 1
```

The *Mode* property of a select determines the appearance of a selected token when these colors are not explicitly specified. Please consult the resource manual for an explanation of this property.

To emphasize the EXP select when it overlaps the current selection, we make use of the ctedit *Priority* property. When two selects overlap, the select assigned the highest priority number wins. By default, all selections, including the current selection, have priority zero, so in our resource specifications, we assign the highlight select a priority of one.

7.8 If that weren't enough...

Whew. The EXP environment should now be up and working. You may use it to test these other CENTAUR tools.

7.8.1 Debugging the interpreter

Recompile the interpreter in debug mode, change the environment variable `#:Exp:interpreter:debug` to `t`, and trace the evaluation of an EXP program with the TYPOL debugger. The TYPOL user's manual gives a detailed example of how the debugger works.

7.8.2 In-line edition

Add in-line edition to the EXP environment so that you may edit the input environment tree directly in the initial environment view. The `centaur-language` command generates everything necessary to do so. You need only edit the file:

```
contrib/Exp/pprinters/pprinters.rdb
```

and change the resource:

```
Centaur.Exp.ppml.std.Package: (env basic)
```

to:

```
Centaur.Exp.ppml.std.Package: (line env basic)
```

Then compile the pretty printer by running `ctmake` in the subdirectory `line`.

You must reset resources and redraw any ctedit containing an EXP program to account for the change. Please consult the edition manual in the Mantle for information about in-line edition.

7.8.3 Program transformation

Add program transformation to the environment. The edition manual in the Mantle describes how to generate and personalize menus that offer language specific program transformations. As in guided editing mode, clicking on a menu entry transforms the selected source text. For the EXP language, such a menu might contain transformations to distribute multiplication, the inverse, etc.

7.8.4 Resources for EXP

Add *resource manager objects* to the EXP environment so that values such as button titles, view names, and the debug flag may be set by resources rather than hardcoded. Resource manager objects (rmo's) allow a tool to interact with CENTAUR's resource database. When defining a tool class (such as the interpreter tool class), it is interesting to define an rmo class alongside. Whereas the tool class defines properties required by the tool to perform its computations, the rmo class defines resource properties required by the tool to access important values.

Once you have defined an rmo class, you may create an rmo instance required for access to the resource database. The object is registered in a tree of rmo instances global to the CENTAUR system. A path in the tree of rmo instances determines the resource path required to specify a value for a property. More precisely, each rmo instance has an associated query that it uses to interrogate the database. The query's list of instance and class names is formed by the instance and class names of the query's ancestors in the rmo tree.

Rmo instances that share the same parent in the rmo tree must differ in either class or instance name. The rmo object must also be accessible to the tool (either stored in it or visible globally) so that at any moment during a computation, the tool may use it to interrogate the resource data base. For complete information about rmo's, please consult the resource manual. The following Le-Lisp code should also be included in `contrib/Exp/environment/interpreter.ll`

The first task, therefore, is to define an appropriate rmo class for the interpreter tool with the `defrmo` function.

```
(defrmo Interpreter
        (Debug symbol none)
        (Initial-view-title string "Initial Env")
        (Result-view-title string "Result Env"))
```

The *Interpreter* class specifies three properties: Debug, Initial-view-title, and Result-view-title. The allowed value type follows each name[19], then the default value. Once we have defined the class, we may create instances with the function `#:rmo:create`. This function takes a class name and an instance name. For example, CENTAUR creates an rmo instance for each formalism object declared by resources by calling this function with the *Formalism* rmo class name and as instance name, the formalism name.

What instance name should we choose for our rmo? The answer to this question depends on the resource semantics we require. If we desire that each tool be able to specify different values for the resources, we must provide each rmo with a distinct instance name. For example, we may want to interpret one EXP program in debug mode and another in normal mode. On the other hand, we may not require such individuality, and the same resources could apply to all tool instances. When dealing with formalisms, for example, we definitely need to be able to specify different resources for different formalism instances. In the EXPenvironment, for example, we probably don't require button titles to vary between interpreter tool instances. In this case, we only require one rmo instance —and therefore one instance name— available to all interpreter tool instances. In this tutorial, no name distinguishes tool instances that we might use as an rmo instance name. We therefore elect

[19] The type `symbol` must be used for packaged symbols.

the global rmo option. Note that were we to create a distinct rmo instance for each tool instance, we could extend the tool class to allow us to store an rmo in the tool. However, since our rmo instance will be unique and global, we will make it available otherwise.

The next important issue is where to register the rmo instance in the rmo tree. This choice depends on the resource path we want to follow in order to specify interpreter properties. The interpreter depends on the EXP formalism and is implemented in TYPOL. It makes sense to specify interpreter properties by some path that encompasses that information. For every declared formalism, CENTAUR creates instances of the predefined *Formalism* and *Semantics* rmo classes, and registers these instances in the rmo tree. If we register the interpreter rmo below the *Semantics* instance, we generate the following query:

```
< <centaur-name>.Exp.semantics.interpreter |
  Centaur.Formalism.Semantics.Interpreter>
```

With this query we may specify interpreter property values, for example, by:

```
Centaur.Exp.Semantics.interpreter.Debug : t
Centaur.Exp.Semantics.interpreter.Initial-view-title : Initial View
Centaur.Exp.Semantics.interpreter.Result-view-title : Final View
```

The following code creates and registers an instance of the *Interpreter* rmo class. If such an rmo has already been created, it is returned.

```
(de :rmo:find ()
    (let ((semantics-rmo (#:SemanticsManager:semantics-rmo 'Exp)))
    (or (#:rmo:subobject semantics-rmo 'interpreter 'Interpreter)
        (let ((new (#:rmo:create 'interpreter 'Interpreter)))
             (#:rmo:register-object semantics-rmo new)
             new))))
```

This function may be called by the interpreter tool wherever it needs to access the database. For example, to determine debug mode:

```
(de :callback:interpret (stnode port)
  (lets ((tool (st-object stnode))
         (root (:tool:get-source tool))
```

```
        (env  (:tool:get-initial-env tool))
        (debug
          (neq 'none (#:properties:get-property (:rmo:find) 'Debug)))
        )
    ;;If no environment tree memorized by tool,
    ;;call appropriate Typol predicate.
    (if env
        (ty-run tool '#:eval_exp:full_eval_exp debug root env)
        (ty-run tool '#:eval_exp:init_eval_exp debug root)
        )))
```

The generic function `#:properties:get-property` takes an rmo and a property name and returns the value stored in the resource database. **N.B.** We may not specify **nil** as a property value, so instead we specify **none** which we then interpret to mean nil.

Two issues remain: Where do we store interpreter resources? And how do we load them automatically into the resource database with the Exp environment? We suggest specifying interpreter resources in a file named:

`contrib/Exp/semantics/interpreter/interpreter.rdb`

The resources provided by an environment must be general enough to allow users to override them, and specific enough to not be subsumed by other system resources. For the Exp environment, it makes sense to use the asterisk up to the formalism name, and classes everywhere else.

```
*Exp.Semantics.Interpreter.Debug : none
*Exp.Semantics.Interpreter.Initial-view-title : Initial View
*Exp.Semantics.Interpreter.Result-view-title : Final View
```

We would like to load resources from this file when necessary, i.e., at the first request for an interpreter property value. To do so, we create an instance of the predefined *Database* rmo class which simply describes the location of a database file. We will register this rmo instance under the *Interpreter* instance, so it should be created at the same time:

```
(de :rmo:find ()
    (let ((semantics-rmo (#:SemanticsManager:semantics-rmo 'Exp)))
    (or (#:rmo:subobject semantics-rmo 'interpreter 'Interpreter)
        (let ((new (#:rmo:create 'interpreter 'Interpreter)))
             (#:rmo:register-object semantics-rmo new)
```

```
;;Create Database rmo and load database.
(#:databaseMan:load (#:databaseMan:database new))
new))))
```

The previous function will load the database file automatically provided some specification identifies the database location when the function is called. Therefore, the following resource should be included in the formalism resource file `Exp.rdb`:

```
*Exp.Semantics.Interpreter.Database.Location : \
contrib/Exp/semantics/interpreter/interpreter.rdb
```

If you modify these resources dynamically, don't forget to reset the database!