The Ppml Manual

Ian Jacobs

The Ppml Manual

Ian Jacob

February 1991

Contents

1 Introduction

An unparser, or pretty printer, translates an abstract syntax tree into text. In Centaur, pretty printer specifications are written in Ppml, compiled into Lisp code, and executed by a the Ppml *formatting machine* in conjunction with an abstract syntax tree. The formatting machine generates text which is displayed in a Centaur editor.

A Ppml pretty printer for a language typically consists of a sequence of rules having the form:

<pattern> → <format>

The formatting machine executes right hand side of a rule, the formatting part, when the name of the node being examined matches the pattern on the left hand side.

Patterns correspond to the names of fixed arity, list, and atomic abstract syntax nodes, usually those defined in Metal or Sdf. In addition to names, Ppml patterns may contain variables which, in case of a match, instantiate to the matched abstract syntax tree node. Ppml also provides several mechanisms that enable pattern matching in specific contexts, according to a desired detail, and according to additional information stored in the abstract syntax tree.

The right hand side of a Ppml rule dictates the formatting of the matched tree node in terms of a box language. Separators determine the layout of terminals specified in these boxes, in other words, the pretty printing of tokens.

Ppml also provides a number of constructs to combine and simplify rules, call external Lisp functions, and to define constants and default values that may be used throughout the program.

This manual discusses all aspects of the Ppml language. In the Users manual, you will find information about the graphic interface and its functionalies, the formatting machine, the default and generic pretty printers, selection behavior, as well as a programming style manual and a list of known bugs. The syntax of the language is described in the Reference manual, complete with examples. Errors are discussed in a separate section, as are functions available in case of need. The concrete syntax of the language as well and the Metal definition have been included as handy references.

2 The User's Manual

2.1 The Ppml environment

When you successfully read, i.e., parse an error free Ppml program in a Centaur editor, Centaur invokes the Ppml environment, which offers the following functionalities:

- a compiler,
- limited navigation facilities,
- limited selection facilities.

The buttons corresponding to these functions may be found either in the *Ppml* pulldown or with the right mouse button when a Centaur editor contains a Ppml file. In the following explanations, we refer to the language being pretty printed as the object language.

2.1.1 Compile

The *Compile* button does the following for a program written in a formalism L:

- If in **compat** mode, this button functions as before, namely, it compiles the program and generates the file `L-pp.ll` in the current directory. This file contains the functions that the formatting machine uses to pretty print abstract syntax trees for the object language. Any compiling errors are displayed in a Centaur error window.
- Otherwise, compiles the program and generates two files: `L-pp.ll` and `module.LM` in the directory specified by the *Root* and *Location* resources for the pretty printer. The `module.LM` file allows you to compile the `.ll` file to improve pretty printer performance.

Please consult the Centaur resource manual for details about resource specifications and the Centaur tools manual for details about compiling pretty printers.

2.1.2 GotoRule

All of the "Goto" buttons described below cause the following to happen:

- A query box prompts you for the pertinent "Goto" information.
- The desired item is selected, unless it doesn't exist, in which case a message is printed in the home CENTAUR window.

The *GotoRule* button selects the first rule headed by the indicated operator name. You may not select constrained patterns via this button.

2.1.3 GoUpRule

If the interior of a rule has been selected, this button selects the entire rule. If no part of a rule has been selected beforehand, the error message "`you are not in a rule`" appears in the home window.

2.1.4 GotoFunction

This button selects the indicated internal function definition, if it exists.

2.1.5 GotoPhylum

This button selects the indicated internal phylum definition, if it exists.

2.1.6 GotoChapter

This button selects the indicated chapter, if it exists.

2.1.7 ShowRules

This button selects all existing rules for the indicated operator, and prints the number of rules in the home window.

2.1.8 ShowContext

This button selects all existing rules in the indicated context.

For information about buttons which change the print level of a displayed program, which pretty printer is used, how to modify selections, etc., please consult "The User Interface Manual."

2.2 The formatting machine

The *formatting machine* pretty prints an abstract syntax tree according to Lisp functions generated by the Ppml compiler for a particular object language. Although you needn't understand the details of these Lisp functions, you should understand some basics about the formatting mechanism and a few Ppml heuristics so that your Ppml programs behave as expected.

2.2.1 The buffer

Life would be *so* easy if the formatter had only to align terminals horizontally. Unfortunately, many programs require more than one line of text, so the formatting machine has to address the issue of when to break lines. The question of vertical alignment means that the formatting machine cannot always print a token at the moment it is read. Therefore, it gradually fills a buffer with tokens and separators and empties the buffer (and prints the contents) whenever possible, in conjunction with the page size. The general principles governing the filling of the buffer are:

- single separators alternate with single terminals, beginning and ending with a terminal:

  `<term> <sep> <term> <sep> ... <term>`

- the type of a box is determined by its global separator, which lies implicitly between all elements of a box. A local separator overrides the global separator for the terminal that follows it. As each box is processed and its tokens fill the buffer, it remembers:
  - which tokens belong to it,

– break point information (see below).

so that the formatting machine can determine when to go to the next line.

Thus, for example, the following formatting part:

```
[<h> "token1" "token2" <v> "token3" "token4"] ;
```

is converted in the buffer to:

```
"token1" <h> "token2" <v> "token3" <h> "token4"
```

Similarly, the instruction:

```
[<h> "token1" [<hv> "token2" "token3"] "token4"] ;
```

is converted in the buffer to:

```
"token1" <h> "token2" <hv> "token3" <h> "token4"
```

Notes

- Empty boxes and the separators that precede them are ignored in the buffer. Thus, the rule:

  ```
  [<h> "token1" [<v> ] "token2"] ;
  ```

  is interpreted in the buffer as:

  ```
  "token1" <h> "token2"
  ```

- "Reckless" program design may produce situations in which a box contains several separators in a row. In this case, the formatting machine considers only the last separator of the group. Thus, the box:

  ```
  [<h> "token1" "token2" <v> <hv> "token3"] ;
  ```

is equivalent to:

```
[<h> "token1" "token2" <hv> "token3"] ;
```

- The formatting machine ignores any local separators at the beginning or end of the buffer, a phenomenon also introduced by lax programming. Thus, the buffer:

```
[<h> <v> "token1" "token2" "token3" <hv>] ;
```

is equivalent to:

```
[<h> "token1" "token2" "token3"] ;
```

2.2.2 Formatting principles

The formatting machine encounters two situations where it may start printing on the next line:[1]

- when it encounters a vertical separator (<v> or <v1>) in the buffer.
- when it verticalizes a conditional separator (<hv> or <hov>) in the buffer.

2.2.2.1 Breaking at a vertical separator

A vertical separator always breaks any kind of box. Processing a vertical separator in the buffer involves the following actions:

- *All* surrounding <hov> boxes in the buffer are verticalized (described below).
- The buffer is emptied up to and including the vertical separator and all terminals are printed.
- The vertical separator determines how many spaces to indent and lines to skip, and formatting continues.

[1] Called "breaking a box."

2.2.2.2 Breaking at a conditional separator

Processing a conditional separator is a more subtle endeavor. As long as the tokens do not extend beyond the end of the page, the formatting machine adds them to the buffer. If overflow occurs, implying that no vertical separator has been found beforehand, the formatting machine backtracks in the buffer and tries to "verticalize" a suitable conditional separator. A conditional separator may be verticalized when:

- it has been issued by a box of the same type. Local conditional separators issued by other types of boxes are converted as described below.
- (for <hv> separators) it is the right most separator in the outer most box. The outer most box corresponds to the highest box in the tree of boxes read while filling the buffer. The formatting machine must therefore keep track of the tree of boxes that has been read in filling the buffer.

In the following rule, for example, all of the separators are either horizontal or conditional:[2]

```
[<h> "token1" "token2" [<hv1> "token3"
     [<hv2> "token4" "token5" ]
     "token6"]];
```

Suppose that the formatting machine, while contentedly filling the buffer:

```
"token1" <h> "token2" <h> "token3" <hv1> "token4" <hv2>
```

realizes that `"token5"` does not fit on the page. It consults the tree of boxes, i.e. the rule and not the buffer, for the first available break point. The formatting machine traverses the tree of boxes, starting from the root, in the direction of the box containing the guilty token, and asks each box if and where it may be broken. Consider our example rule:

The <h> box, as a horizontal box, answers "I cannot be broken by a conditional separator."
(The formatting machine continues its quest...)

[2]Subscripts in the <hv> boxes are there strictly for reasons of clarification.

The <hv> box answers "I can be broken at the terminal `"token3"` since the separator between `"token3"` and `"token4"` is a conditional separator come from a conditional box."

The terminal `"token4"` is thus printed the default number of lines below `"token3"`, indented by the default number of spaces. The buffer is emptied up to `"token4"`, and formatting continues with a new buffer containing first `"token4"` and then as follows:

```
"token4" <hv> "token5" <h> "token6" ...
```

Closing a box does not clear the buffer, but it does henceforth prevent the formatting machine from breaking the box.

A few remarks:

- The formatting machine would not have considered a local <hv> separator between `"token1"` and `"token2"` as a valid break point since local conditional separators may never break a horizontal or vertical box.
- Similarly, a local <h> separator between `"token3"` and the $<hv_2>$ box:

  ```
  [...  "token2" [<hv1> "token3" <h> [<hv2> ...  ]  ;
  ```

  would have eliminated the chosen break point as a candidate. In this case, the formatting machine would have examined the $<hv_2>$ box and cut between terminals `"token4"` and `"token5"`.
- If no valid break point had been found, the formatting machine would have continued printing the terminals horizontally, even far beyond the specified page width. A box may never be broken at a horizontal separator.
- Breaking a <hov> box is equivalent to changing the box into a <v> box.

In conclusion, keep in mind the following facts:

- the formatting machine converts a <hv> in a <h> or <v> box into a <v>.
- the formatting machine converts a <hov> in a <h>, <v>, or <hv> box into a <v>.

Notes

Don't rely on these conversions as they exist not as shortcuts but for the sake of consistency. Please consult the reference manual for the complete table of conversions.

2.3 Pretty printer issues

2.3.1 Dependence on tables

Since the Ppml compiler uses information stored in the tables of a language (the `.t` file), it is necessary to recompile a Ppml program after recompiling a Metal or Sdf program.

2.3.2 The default and generic pretty printers

When the formatting machine does not find a rule in a pretty printer for a given abstract syntax operator, it invokes the *generic* pretty printer. This pretty printer displays nodes using the following syntaxes:

fixed arity operators: The operator's name is followed by parentheses containing the pretty printed descendents.

list operators: The operator's name is followed by brackets containing the pretty printed descendents.

atomic operators: Only the value of the operator is printed.

The generic pretty printer indicates that a node's print level is zero as follows:

fixed arity operators: #

list operators: ...

atomic operators: the value of the operator.

The formatter does not print nodes whose print level is less than zero.

Since the generic debugger syntax often proves useful for debugging, Centaur offers the *default* pretty printer. Unlike the generic pretty printer, this one may be invoked explicitly. The default pretty printer syntax differs from that of the generic pretty printer only slightly: the name of an atomic operator is followed by its value.

The default pretty printer prints the value of annotations when the variable `{decompilo}:with-annotations` is set to true. This variable is true by default. The generic pretty printer, on the other The generic pretty printer, on the other hand, does not print the value of annotations.

When you create a Centaur editor, the pretty printer is automatically set to `std`. Centaur invokes the generic pretty printer if you try to read a program in the object language when the `std` pretty printer does not exist.

When you read a program into the editor, Centaur loads the Lisp functions for the pretty printer used, and adds its name to the list of known pretty printers. Whenever a file in the object language is read, Centaur invokes the first applicable pretty printer in the list, so to make a pretty printer invisible, you must remove it from the list with the command:

```
({centaur}:remove-pprinter 'language_name 'pp_name)
```

2.3.3 The mouse

When you click on a token in a Centaur window, you select the node in the abstract syntax tree that was matched by the rule that printed the token. Consider the following simplified excerpt from a Pascal-like language which we call `L`:

```
...

program(*ident, *progbody)  ->
   [<v> *decl_s "begin" <v> *instr_s "end"] ;

decl_s(**decl_s) -> [<v> (**decl_s)] ;

var_sdecl(*ident_s, *type, *expr) ->
```

```
   [<hv> *ident_s ":" *type ":=" *expr ";" ] ;

ident_s(*ident1, **ident) ->
   [<hv> *ident1 (<h> "," **ident)];
...
```

which concerns the pretty printing of variable declarations. Suppose that this pretty printer displays a piece of program written in `L` as follows:

```
      ...
   x,y : integer := 0 ;
   t,f : boolean ;
   z : real ;

   begin
      ...
```

The mouse click would behave as follows:

- Clicking on the assignment operator (":=") selects the node `var_sdecl` since the operator was printed by this rule. As a result, the entire declaration **x,y : integer := 0 ;** would be highlighted.
- Clicking on the colon (":") or semi-colon (";") in any declaration selects the node `var_sdecl` for the same reason. Note that clicking cancels the previous current selection.
- Clicking on the variable `x`, dragging the mouse over a colon and releasing selects the node `var_sdecl` since it is the highest node in the tree among those selected.
- Clicking on a colon, dragging the mouse over the word `begin` and releasing selects the entire program since the node `program` is higher up the tree than `var_sdecl`.
- Clicking on any identifier selects the rule that pretty printed the identifer. Note that this is not the rule `ident_s`, which pretty prints a list of identifiers.

- Clicking on the comma between identifiers does nothing because it is printed, in the rule `rule_s`, within the iterator construct.
- The *Clear All* button in the *Selections* pulldown erases all selections, current, error, and otherwise.

2.4 Style manual

As with any programming language, there are a certain number of stylistic principles for PPML which we enumerate below:

- The following suggestions involve reducing and centralizing code:
  - Group rules into chapters.
  - Group operators into phyla.
  - Use constants and default values.
  - Use `where` since it results in more efficient code.
  - Use external function calls for sophisticated operations such as dressing up a terminal or manipulating a tree.
- PPML rules should be compatible with the parser so that edition in the CENTAUR editor corresponds to the parser's recognition of tokens. All trees that may be copied to the clipboard, for example, should be parsable.
- Avoid putting blanks around terminals: `" terminal "`. You may accomplish the same thing by using global separators with parameters and overriding the spacing values with local separators when necessary.
- Staggered pattern matching allows you to play with the mouse selection. For example, the following rule:

```
label_node(label(*label),*exp) →
      [<h> integerpp(*label) *exp] ;
```

  eliminates the possibility of selecting the node `label`. Clicking on the `*label` selects the node `label_node` since `*label` is printed by this rule.

- The following suggestions concern the development of a PPML program:
  - Work incrementally. Write a few rules and test them interactively.
  - Don't work rule by rule but rather by groups of rules (eg.: all of the binary operators of a language at once).
  - Don't worry about print levels.
  - Don't worry about blanks. You can repair small errors later.
  - Try to write the fewest number of rules possible, but stay simple. You should almost always be able to write fewer rules than the number of operators in the language by merging rules with `where`, `case`, etc.
  - Avoid catch all rules. They hide bugs.
  - Verify mouse behavior along the way. Make sure that you select what you think you are selecting.

2.5 Known bugs

- Although you may use `case` in the iterator, you may not use `if`.
- Nested annotations (annotations in trees that are annotations) are bugged.
- Although the compiler is capable of detecting more than one error in a program, it is not capable of detecting more than one in the same rule.
- You can't edit comments.
- The compiler's warning *No rule for an operator* may be masked by a print level rule such as:

  ```
  *x !0 → [<h> "#"] ;
  ```

  This is not a catch all rule; it only applies when the print level is less than or equal to zero. The compiler does not recognize that for any other print level, no rule will apply for the given operator.

3 Reference manual

3.1 Manual structure and notation

In this document, we present the Ppml language in its entirety. We have grouped important concepts together into sections, in which we discuss:

Syntax The complete syntax, in BNF-like notation, of the pertinent language structures.

Description A description of the concept in question.

Example(s) One or more examples of usage.

Notes Important or interesting addenda.

We use the following notation throughout the manual:

=	is defined as
|	or
[x]	0 or 1 occurrences of x
{x}*	0 or more occurrences of x
{x}+	1 or more occurrences of x
(x | y)	x or y

the following typesetting conventions in the syntax definitions for Ppml:

- *non-terminals*
- `terminals`
- "metasymbols" (symbols reserved by Ppml between quotes)

and the following terminology:

- The language being described in Ppml is called the *object language*.

3.2 Lexical elements

The following sections describe the various classes of lexemes found in the syntax definitions of Ppml.

3.2.1 Special symbols

Special symbols include PPML keywords:

and	case	chapter	constant	default	elsif
else	end	function	if	in	is
not	of	or	others	phyla	prettyprinter
rules	term	then	where	with	

or any of the following symbols:

+	-	;	,	::	:
<	>	>=	<=	=	→
[	]	{	}	(	)
↑	*	**	!	\	

3.2.2 Integers

Syntax

$$\begin{aligned} \textit{integer} &= \{\textit{digit}\}+ \\ \textit{digit} &= \textit{0} \mid \textit{1} \mid \ldots \mid \textit{9} \end{aligned}$$

3.2.3 Identifiers

Syntax

$$\begin{aligned} \textit{identifier} &= [\text{“ \texttt{\#}”}]\ \textit{letter}\ \{\textit{alphanum}\}* \\ \textit{alphanum} &= \textit{wordsep}\ (\textit{letter} \mid \textit{digit}) \\ \textit{wordsep} &= \text{“ \_”} \mid \text{“ -”} \\ \textit{letter} &= \textit{uppercase} \mid \textit{lowercase} \\ \textit{lowercase} &= \textit{a} \mid \textit{b} \mid \textit{c} \mid \ldots \mid \textit{z} \\ \textit{uppercase} &= \textit{A} \mid \textit{B} \mid \textit{C} \mid \ldots \mid \textit{Z} \end{aligned}$$

Examples

```
SmallTab
statement_list
#if
```

Notes

If a PPML keyword is used as an identifier, it must be prefixed by a sharp symbol ("#").

3.2.4 Strings

Syntax

$$
\begin{array}{rcl}
\textit{string} & = & \text{“ ’”}\{\textit{string\_elem}\}*\text{“ ’”} \\
\textit{string\_elem} & = & \textit{quote\_image} \mid \textit{string\_character} \\
\textit{quote\_image} & = & \text{“ ’’”} \\
\textit{string\_character} & = & \textit{any\_character\_except\_CR\_or\_quote}
\end{array}
$$

Examples

```
'this is a string'
'A'
''''
```

3.2.5 Terminals

Syntax

$$
\begin{array}{rcl}
\textit{terminal} & = & \text{“ "”}\ \{\textit{terminal\_elem}\}*\text{“ "”} \\
\textit{terminal\_elem} & = & \textit{repeated\_quote\_image} \mid \textit{terminal\_character} \\
\textit{repeated\_quote\_image} & = & \text{“ ""”} \\
\textit{terminal\_character} & = & \textit{any\_character\_except\_CR\_or\_repeated\_quote}
\end{array}
$$

Examples

```
"if"
":"
""""
```

3.2.6 Comments

Syntax

Comments begin with the double minus sign ("--") anywhere on the line and continue to the end of the line.

3.3 Rules

Syntax

rule_part = " `rules`" *rule_list*
rule_list = {*rule*}+
rule = *pattern* " →" *formatting_part* " ;"

Description

The *rules* of a Ppml program describe the textual formatting of abstract syntax operators. The right hand side of a rule is executed when an abstract syntax tree node[3] matches the pattern on the left hand side of the rule. When pattern matching succeeds, i.e., the rule "applies" to the node, the right hand side is executed using the values of all variables instantiated in the pattern.

Notes

- Rule order is important. Rules are applied in the order they appear, so the most specific rules should appear at the beginning of a program, followed by more and more general rules. Be carefule about sweeping rule such as:

  $$*\texttt{x} \rightarrow \textit{formatting\_part}$$

  which masks all successive rules.

- It is possible to define several rules for the same operator, the most specific appearing earliest in the Ppml program.

- If no rule applies for a given operator, the formatting machine invokes the generic pretty printer.[4]

[3] and context: see the sections on print level and annotation matching.
[4] See the User's Manual for details.

3.3.1 Patterns

Syntax

pattern = *simple_patt* | *constraint_patt* | *context_patt*
| *annotated_patt* | *printlevel_patt*

Description

PPML provides several sophisticated pattern matching techniques for precise control over the mapping of an abstract syntax tree into text.

3.3.1.1 Simple patterns

Syntax

simple_patt = *variable*
| *op_name* [" (" [*pattern_list*] ")"]
| *op_name atomarg* | *op_name*
op_name = *identifier*
atomarg = *variable* | *string*
variable = *anonymous_var* | *named_var*
anonymous_var = " ∗"
named_var = *general_var* | *list_var*
general_var = " ∗" *var_name*
list_var = " ∗∗" *var_name*
var_name = *identifier*
pattern_list = *simple_pattern* {" ," *simple_pattern*}∗

Description

Simple patterns may be variables, specific operator names, or a combination of the two.

There are three kinds of variables:

general: A general variable, designated by ∗`name`, matches any operator in the abstract syntax tree, and instantiates to the value of the current node.

anonymous: An anonymous variable, designated by $*$, matches any operator, but without instantiation, i.e., the value of the current tree is lost.

list: A list variable, designated by $**$`name`, matches the descendents of a list operator and instantiates to the list.

Example *Variable names*

```
*statement
*decl
**decl_s
*
```

Notes

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

- A variable name found on the right hand side of a rule must appear on the left.

- You may not use a variable name more than once in a pattern. The following patterns are therefore illegal:

```
op(*x,*x)
```

```
op(*x,**x)
```

Since Metal and Sdf support fixed arity, list, and atomic abstract syntax nodes, Ppml provides a pattern matching mechanism for each case.

3.3.1.1.1 Fixed arity patterns.

A fixed arity pattern applies when the text of the pattern and the number of descendents exactly matches the name and arity of an abstract syntax node. Descendents may be general variables, anonymous variables, and operator names. For example, the pattern:

```
oper(*x, *y)
```

matches a node named `oper`, and assigns the two descendents to the variables `*x` and `*y` to be used during formatting. Note that this pattern differs from:

```
oper(x, *y)
```

which matches a node named `oper` whose first descendent must be the operator `x`. PPML accepts nested patterns such as:

```
oper(oper2(*x), *y, oper3(oper4(*z)))
```

Notes

Fixed arity operators may not have list variable descendents. The following pattern for the binary operator `plus` is illegal:

```
plus(*exp,**exps)
```

3.3.1.1.2 List patterns

A list pattern applies when the text of the pattern matches the name of a list node and the list of elements being pretty printed is compatible with the number of descendents in the pattern. Descendents may be general variables, anonymous variables, and operator names. A list pattern with several descendents implies a decomposition of the list. For example, the pattern:

```
oper_s()
```

matches a list node named `oper_s` that has no descendents. The pattern:

```
oper_s(**x)
```

matches a list node named `oper_s` that has any number of descendents. The list of descendents is assigned to the list variable `**x`.
You may break down a list using general and list variables, as in:

```
oper_s(*x, **y, *z)
```

which assigns `*x` the first element in the list, `*z` the last element in the list, and `**y` the rest of the elements in the list.[5] Note that for such a pattern to apply, the initial list of descendents must contain at least two elements, corresponding to `*x` and `*z`.

You may use a list variable only **once** in a list pattern. More than one introduces ambiguity about where to divide up the list. The following pattern is therefore illegal:

```
oper_s(**x, **y)
```

Patterns for list nodes that don't contain a list variable match only those lists with exactly as many elements as descendents in the pattern. The pattern:

```
oper_s(*x, *y)
```

thus matches the node `oper_s` when it's descendent list contains only two elements.

3.3.1.1.3 Atomic patterns

An atomic pattern applies when the text of the pattern matches the name of an atomic node. The variable following the node name instantiates to the node's value. Thus, the pattern:

```
ident *x
```

matches an atomic node named `ident` and assigns the value of `ident` to the variable `*x`. Note that no parentheses follow the atomic node name, indicated a zero arity node. It is possible to limit the matching of atomic nodes to specific values by substituting a string for the variable part. Thus:

[5] Do not mistake this pattern for that of a node with three descendents.

```
ident 'myident'
```

applies when the value of `ident` equals the string `myident`.[6]

Notes

- A pattern designated for a singleton node must consist only of the operator name. No variable or string may follow since singletons have no values attached.

3.3.1.2 Constrained patterns

Syntax

constraint_patt = *simple_patt* " `where`" *cond_list*
cond_list = *cond* {" `,`" *cond*}∗
cond = *bool_exp*
bool_exp = *arith_exp* (" =" | " >" | " >=" | " <" | " <=") *arith_exp*
| " `not`" " `(`" *bool_exp* " `)`"
| *bool_exp* (" `and`" | " `or`") *bool_exp*
| *var_list* " `in`" *phylum_spec*
var_list = *named_var* {" `,`" *named_var*}∗

Description

Constrained patterns behave like simple patterns, but offer a mechanism to limit the scope of variables to specific operators. For example, the pattern:

```
*x where *x in {op1, op2, op3}
```

matches only those operators named `op1`, `op2`, or `op3`. After instantiation, the variable `*x` has as its value the operator selected from the pattern list {op1, op2, op3}. In the pattern:

```
*x where *x in {op1(*a,*b),op2(*a,*b)}
```

[6]String pattern matching only works for those atomic values internally represented as strings, namely STRING, IDENTIFIER, and CHAR, and integers, namely INTEGER.

`*x` instantiates to the value of the current tree, and `*a` and `*b` to its descendents. You may thus manipulate the current tree and its descendents in the same rule. [7]

You may also used constrained patterns for variables appearing within another pattern. The pattern:

```
op1(*x,*y,*z) where *x,*y in {op2},*z in {op3}
```

matches a tree named `op1` when the first two descendents are `op2` and the third is `op3`.

The pattern:

```
op1(**x) where **x in {op2, op3}
```

matches a node named `op1` when every operator in the list of descendents is either `op2` or `op3`.

You may also replace the pattern list (the part in curly braces) by a phylum name, from either the abstract syntax definition of the object language or one defined in PPML.[8] Thus, in a pattern such as:

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

`*x` matches and instantiates to any operator belonging to the phylum EXP. It is even possible to perform union and difference operations on phyla such as:

```
*x where *x in EXP - {div, minus}
```

or, if OPS is another phylum:

```
*x where *x in EXP - OPS
```

Finally, it is possible to constrain a pattern with boolean expressions, as in:

```
op(*x) where precedence(*x) > 1
```

[7] See "A Centaur Tutorial" for a good example of when this might be useful.

[8] See the section on PPML phyla definitions for details about local phylum definitions.

or:

```
*x where *x in {op(*y,*z)},
       precedence(*x) < precedence(*y) or
       precedence(*x) > precedence(*z)
```

This last pattern illustrates how intricate PPML patterns may become. We encourage you, however, to simplify rules as much as possible.

Notes

All operators in a pattern list must:

- have the same arity,
- use the same names for and number of variables in the pattern. The following pattern is thus illegal for two reasons:

```
*x where *x in op(*y,**z),op(*y,*wrongvar),wrongop(*y)
```

3.3.1.3 Print level patterns

Syntax

printlevel_patt	=	(*simple_patt* | *constraint_patt*) " ! " *printlevel*
printlevel	=	*printlevel_name* | *integer*
printlevel_name	=	*identifier*

The print level mechanism allows you to control the detail of a program displayed in a CENTAUR editor according to the depth of a node in the abstract syntax tree. Suppose that the following pattern matches the node `op`, whose print level is 8:

```
op(*x,op1(*y))
```

By default, the PPML formatting machine decrements the print level one for each tree level. The immediate descendent `*x` therefore has a print level of 7. Since the variable `*y` is yet another level farther down the tree, its print level is 6. The print levels of subsequent descendents of `*x` and `*y` decrease until either the print level reaches zero or recursive calls cease. The default and generic pretty printers indicate a print level of zero with a special symbol

(for example, $***$ in this version of CENTAUR). Nodes in the tree "lower" than a node whose print level is zero are not displayed.

It is possible to invoke a PPML rule for specific print level values. The pattern:

```
*x !5
```

matches any tree node whose print level is between 0 and 5 inclusive. You may have noticed that the syntax of print level patterns resembles that of atomic patterns:

```
node_pattern value
```

As with atomic patterns, it is possible, when the pattern matches, to assign the value of the print level to a variable to be used in the formatting part of the rule. For example, the pattern:

```
*x !prlevel
```

assigns the current print level to the variable `prlevel`.

Notes

The notation `!n` means less than or equal to `n`, which means that if you have two rules such as:

```
*x !0 -> ...
*x !1 -> ...
```

the first will apply for any print level less than zero and the second only when the print level equals one, since in all other cases the first rule renders the second invisible. Reversing the rules, on the other hand:

```
*x !1 -> ...
*x !0 -> ...
```

causes the first rule to obliterate the second in all cases.

3.3.1.4 Annotation patterns

Syntax

annotated_patt	=	*(simple_patt* | *constraint_patt* | *printlevel_patt)* *frame_list*
frame_list	=	{*frame*}∗
frame	=	"↑" *frame_name*
frame_name	=	*identifier*

It is not always possible to store information concerning an abstract syntax tree node as a descendent of the node. Consider comments. They may appear anywhere in a program and must be parsable at any syntactic moment. Without annotations to store comment information, we would have to add the descendent `comment` to every operator in the abstract syntax definition. Centaur offers the annotation mechanism for the treatment of comments and other "non-structural" information, i.e., information outside of the abstract syntax definition. Ppml provides a mechanism to extract this information and format it alongside structural information.[9]

Example *Comments*

Centaur stores comments encountered during parsing in one of two reserved annotations:*prefix* and *postfix*.[10] The annotation patterns:

```
*x ^postfix
*x ^prefix
```

match any nodes annotated with comments. Note that the annotation pattern syntax resembles that of print level and atomic operator patterns, i.e. a pattern followed by a value. As expected, the variable ∗`x` instantiates to the current tree node and the variable ↑*prefix* or ↑*postfix* is assigned the value of the named annotation.

The annotated pattern matching technique has one side effect: Matching a node with an annotation pattern renders the annotation invisible for recursive calls on the node. This prevents you from repeatedly extracting the

[9]See the Vtp manual for details about annotations.

[10]depending on whether the comment precedes or follows the nearest abstract syntax tree node.

same annotation information. All other annotations remain untouched. In short, the rules:

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

- extract the pertinent annotation information (which is stored in either ↑`prefix` or ↑`postfix`),
- evaluate recursively ∗`x` having stripped it of the recognized annotation,
- evaluate recursively either ↑`prefix` or ↑`postfix`.

Note that the last action implies that other rules are often necessary to treat the value of the annotation. Comment annotations have trees as values whose operators are named `comment_s` and `comment`[11] To handle these operators, the METAL pretty printer includes the rules:

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

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

Other annotations might contain trees from other formalisms that require a pretty printer for that formalism. We discuss multiple pretty printing in the section on initializations.

3.3.2 Formatting

Syntax

formatting_part	=	*box*
elem	=	*terminal* | *recursive_call* | *box*
		| *if* | *case* | *sublist_iterator*
		| *ext_func_call* | *local_sep*

[11] These operators are generated automatically by the METAL and SDF compilers for all languages.

Description

The *formatting part* of a PPML rule specifies the textual representation of nodes in the abstract syntax tree. PPML formatting is based on a *box language* which consists of three basic entities:

terminals: which the pretty printer displays,

separators: which specify the arrangement of terminals,

boxes: which organize terminals for issues of:

- cutpoints for the *conditional separators* <hv> and <hov>.
- indentation for the <v>, <v1>, <hv>, and <hov> separators.

3.3.2.1 Terminals

Terminals are described in the section on PPML's lexical elements.

3.3.2.2 Separators

Syntax

$$
\begin{array}{rcl}
\textit{separator} & = & \text{“ <” } \textit{type\_name } [\textit{param\_list}] \text{ “ >”} \\
\textit{default\_sep} & = & \text{“ <” } \textit{type\_name param\_list } \text{“ >”} \\
\textit{param\_list} & = & \textit{param } \{\text{“ ,” } \textit{param}\}* \\
\textit{param} & = & \textit{integer} \mid \textit{identifier}
\end{array}
$$

Description

A separator aligns the elements on either side of it according to the type of the separator. Each separator has optional parameters that describe the horizontal and vertical spacing between elements. If the parameters are not indicated explicitly, the default parameters are used.[12] A separator is said to be "issued" by a box, i.e., a separator is always linked to a box.

There are five types of separators:

<**h** d_x> separates elements by d_x spaces.

[12] See the section on initialisations for information about modifying default values.

<**v** d_i,d_y> separates elements by d_y blank lines and begins printing the right hand element after an indentation of d_i spaces with respect to the left hand margin of the issuing box.

<**v1** d_i,d_y> behaves like <v d_i,d_y> except that the the indentation of the right hand element depends on the left hand margin of the first box surrounding the issuing box. A <v1> separator in anything but a <v1> box is transformed into a <v> separator.

<**hv** d_x,d_i,d_y> separates elements in a box by d_x spaces. If the elements in the issuing box don't fit on the same line, a <hv> separator may serve as a valid cut point for the formatting machine. In this case, <hv> behaves like <v>.

<**hov** d_x,d_i,d_y> separates elements in a box by d_x spaces. If the elements in the issuing box don't fit on the same line, a <hov> separator may serve as a valid cut point for the formatting machine. In this case, all <hov> separators are transformed into <v> separators, and elements following them are stacked with a vertical spacing of d_y. All elements after the first are indented by d_i spaces with respect to the left hand margin of the issuing box.

See the next section on boxes for examples of separator usage.

Please consult the User's Manual section on the formatting machine for details about cut points and separator behavior.

Notes

- Negative indentation is no longer permitted.

3.3.2.3 Boxes

Syntax

$$
\begin{array}{rcl}
\textit{box} & = & \text{“ [” } \textit{global\_sep } \textit{elem\_list } \text{“]”} \\
\textit{elem\_list} & = & \{\textit{elem}\}* \\
\textit{global\_sep} & = & \textit{separator} \\
\textit{local\_sep} & = & \textit{separator}
\end{array}
$$

Description

A box consists of a *global separator* (the type of the box) and a possibly empty list of elements, separated either implicitly by the global separator of the box, or explicitly by an optional *local separator*. A local separator overrides the global separator for the terminal that directly follows.

Here are some examples of formatting parts followed by their behavior and results. We have enclosed results in a box for reasons of clarity — it does not actually appear in a CENTAUR editor. Spaces and indentation are represented by the symbol "␣".

Example

```
[<h 1> "token1" "token2" "token3"] ;
```

aligns the three terminals horizontally, separated by one space:

token1␣token2␣token3

The formatting machine prints the three tokens even if they extend beyond the page width.[13] If you want to change the spacing parameter for one of the tokens, you can override the global separator with the same type of local separator:

```
[<h 1> "token1" "token2" <h bigspace> "token3"] ;
```

which for `bigspace` equal to three, gives:

token1␣token2␣␣␣token3

Example

[13] "Page width" refers to either the default page width or width specified through the CENTAUR editor interface.

```
[<h 1> "token1" "token2" <v 3,1> "token3"] ;
```

aligns the first two tokens horizontally and the third vertically indented three spaces with respect to the left margin of the issuing box (the <h 1> box):

Example

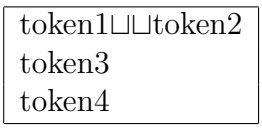
```
[<v> "token1" "token2" "token3"] ;
```

produces a stack of terminals separated by a blank line, and no indentation (corresponding to default parameters 0,1).

Example

```
[<v> "token1" <h 2> "token2" "token3" "token4"] ;
```

separates the first two tokens with an <h> separator. The last two tokens are stacked with no indentation and no line skips (both default values assumed to be zero here).



Example

```
[<h 1> "token1" [<v 3,0> "token2" "token3"] "token4"] ;
```

separates `"token1"` and `"token2"` by one space, followed vertically by `"token3"`, after indenting three spaces with respect to the left hand margin of the current box, or the beginning of `"token2"`:

```
token1␣token2
      ␣␣␣token3␣token4
```

If we replace the <v> box with a <v1> box, the formatting machine indents three spaces starting at the left margin of the surrounding <h> box, or the beginning of `"token1"`.

```
token1␣token2
␣␣␣token3␣token4
```

In this example, we say the global horizontal separator separates `"token1"` from the internal box, and the internal box from `"token4"`. Described more precisely, the global horizontal separator separates the `"token1"` from the terminal immediately to the right (`"token2"` of the internal box), and `"token4"` from the terminal immediately to the left (`"token3"` of the internal box).

Example

```
[<hv 3,0,0> "token1" "token2" "token3" "token4"] ;
```

arranges the four tokens horizontally as long as they fit within the current page width. If they don't, they are folded at the rightmost valid cut point. Suppose that `"token4"` doesn't fit on the page. This formatting part produces:

Example

```
[<hov 1,2,0> "token1" "token2" "token3" "token4"] ;
```

arranges the four tokens horizontally as long as they fit within the current page width. If they don't, they are aligned vertically and indented two spaces:

Notes

- A local conditional separator should only appear in a box with the same type of global conditional separator. If not, it is transformed as indicated below.
- For reasons of compatibility, local separators in certain types of boxes are transformed by the compiler as follows:

global sep →		*h*	*v*	*v1*	*hv*	*hov*
local sep ↓	*h*	*h*	*h*	*h*	*h*	*h*
	v	*v*	*v*	*v1*	*v*	*v*
	v1	*v*	*v*	*v1*	*v*	*v*
	hv	*v*	*v*	*v1*	*hv*	*hov*
	hov	*v*	*v*	*v1*	*v*	*hov*

This being the case, it may be necessary to rewrite rules of honorable intention that are doomed to be transformed. For example, instead of writing the formatting instruction:

```
[<h> "token1" "token2" <hv> "token3" "token4"] ;
```

which will not behave as desired since the <hv> will be systematically converted to a <v>, consider the following right hand side which accomplishes the desired task:

```
[<hv> "token1" <h> "token2" "token3" <h> "token4"] ;
```

- A local <v> separator causes the immediate and irreversible verticalisation of **all** surrounding <hov> boxes. The following rule is therefore useless:

```
[<hov> "token1" "token2" <v> "token4"] ;
```

3.3.2.4 Recursive calls

Syntax

recursive_call	=	[*context_name* " ::"] *simple_call*
		[" !" *printlevel_exp*]
simple_call	=	*variable* | *frame*
printlevel_exp	=	[*printlevel_name*] (" +" | " -") *integer*
		| *printlevel_name* | *integer*

Description

A named variable or a frame (cf. annotated patterns) on the right hand side of a PPML rule represents a recursive call on its value.

Example *Recursive call*

The instruction:

```
[<h 1> *exp1 "+" *exp2] ;
```

prints horizontally the result of the pretty printing of ∗exp1, the plus sign, and the result of the pretty printing of ∗exp2.

Just as it was possible to incorporate the print level of an operator on left hand side of a rule, it is possible to modify the print level on the right hand side. An exclamation point followed by a[14] a plus or minus sign modifies the matched print level for the element that precedes it. The rule:

```
op1(∗x) → [<h> ∗x !+ 1] ;
```

prints the variable ∗x at the same print level as op1 since the instruction +1 cancels the effect of the default mechanism.

An exclamation pint followed by an integer or constant prints the element that precedes it at the indicated print level. Thus, the rule:

```
op1(∗x) → [<h> ∗x !2] ;
```

always prints ∗x with a print level of two.

3.3.2.5 External function calls and Atomic trees

Syntax

ext_func_call	=	*ext_func_name* " (" *arg* ") "
ext_func_name	=	*identifier*
arg	=	*named_var* | *string* | *frame*

Description

An atomic value that is not a tree must be formatted with one of the following :

- the generic pretty printer,
- a predefined function,
- a user-defined external function.

[14] followed by an optional print level name

The generic pretty printer is invoked when no rule applies to an atomic operator. It prints the value of the atomic operator only.[15]

The following predefined functions treat the standard atomic values:

symbolpp for a Lisp symbol,

integerpp for an integer,

characterpp for a character,

stringpp for a string,

bignumpp for a Lisp big number,

identitypp returns its argument if the argument is any kind of atom, else signals an error.

metavariablepp returns its argument prefixed by a dollar sign ("$").

Examples

```
int *x → [<h> integerpp(*x)] ;
ident *x → [<h> stringpp(*x)] ;
*x ↑myannot → [<v> *x stringpp(↑myannot) ] ;
```

You may exert greater control over the pretty printing of atomic values through externally defined Lisp functions. These functions should be stored in a file called:

`language-prettyprinter.at`

where `language` is the name of the object language and `prettyprinter` is the name of a pretty printer for `language`.

These functions must obey the following Lisp package naming scheme:

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

[15] See the User's Manual for details.

where *language* is the name of the object language, **prettyprinter** is the name of the pertinent pretty printer, and `myfunction` is the desired name of the function.

External functions must take a single argument (a variable, string, or frame) and return a string to be printed in the function call's position in the rule.

Example *Atomic values*

One common application of external functions concerns treating individual cases of atomic values. Suppose that a language differentiates between keywords and ordinary identifiers with a special symbol, such as a sharp sign ("#"). If in the `.tokens` file for the language, you specify that the special symbol should be removed at the construction of an abstract syntax tree, the pretty printer for the language should reinsert the symbol so that object language trees may be directly reparsed.[16] Although in PPML you may use pattern matching in a case statement for such fine tuned formatting, an external function provides an easier, more flexible solution. The following imaginary rule for the `std` pretty printer of `mylanguage` shows at what moment the external function is called:

```
ident *x → [<h> ident(*x)] ;
```

The `mylanguage-std.at` file should include functions resembling:

```
(de #:ppmylanguagestd:atm:ident (x)
   (treat_keyword x)
)

(de treat_keyword (symbol)
   (if (memq symbol '(program is end use end chapter block var begin))
       (catenate "\#" symbol)
       (string symbol)
   )
)
```

[16] You might want to remove the special symbol from an identifier in the abstract tree so that any treatment of these values will be systematic and not have to handle the special symbol.

Notes

Recursive calls on atomic values that are not trees provoke a fatal error during the execution of the Ppml program.

Example *Non-atomic trees*

External functions may also pretty print non-atomic trees. One common use involves the reduction of several "similar" rules into a compact rule. Similar rules are those whose elements have the same positions in the same type of box. For example, the following rules for binary arithmetic operators:

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

are identical except for the token between the expressions. Unifying these rules consists of exploiting the `where` construct to create one generic binary operator rule in which the sole terminal is pretty printed by an external function call. The function, which we have chosen to call `getsymbol` selects the appropriate terminal according to the name of the binary operator. Here is the new Ppml rule:

```
*x where *x in {plus(*exp1,*exp2),minus(*exp1,*exp2),
                prod(*exp1,*exp2),div(*exp1,*exp2)}
    -> [<hv> *exp1 getsymbol(*x) *exp2 ] ;
```

and the function `getsymbol` found in associated `Exp-std.at` file:

```
(de #:ppExpstd:atm:getsymbol (tree)
    (selectq (#:operator:name (#:tree:operator tree))
     (plus "+")
     (minus "-")
     (prod "*")
             (div "/")
    )
)
```

Note that this function makes use of VTP primitives to identify the operator name of the `tree` argument.

This combination of `where` and an external function call not only centralizes information in the `.at` file, but results in fewer rules, fewer errors, and more efficient code.

Notes

- External functions must return a **string**.

3.3.2.6 Treating lists

Syntax

sublist_operator = [" \"]" (" *elem_list* ")"

Description

PPML provides an "iterator" to conveniently format list elements with optional separators or terminators. A list variable must appear within the parentheses of the iterator construct, but it may be accompanied by local separators, terminals, general variables, case statements, functions, etc. for special treatment. You should think of this list variable as "each element of the list" so that a rule such as:

```
[<h> (**elems ";")] ;
```

means "each element followed by a semi-colon." Since no local separator appears in the iterator, the global separator controls the formatting. In PPML, lists are not treated as in Prolog, i.e., there are no recursive calls on the "tail" of the list. The PPML iterator linearizes the elements of the list, so that, for example, formatting a list of three elements with the rule:

```
[<h> (**elems)] ;
```

is equivalent to formatting each element "explicitly" with the rule:

```
[<h> elem1 elem2 elem3 ];
```

and the rule:

```
[<h> (**elems ";")] ;
```

is equivalent to:

```
[<h> elem1 ";" elem2 ";" elem3 ";"];
```

The iterator construct functions only as long as the list of elements is not empty. Thus, the preceding rule does not print a semi-colon after an empty list. A backslash ("\") in front of the iterator construct reverses the order of the list elements. Thus, the rule:

```
[<h> \(**elems ";")] ;
```

for a list of three elements, is equivalent to:

```
[<h> elem3 ";" elem2 ";" elem1 ";"];
```

Note that the rule:

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

does not format elements like this:

$elem_1$;
$elem_2$;
...
$elem_n$;

but like this:

$elem_1$
;
$elem_2$
;
...
$elem_n$
;

since the global separator is visible within the iterator. The iterator does not imply grouping of any kind. To print a list of elements each terminated by a semi-colon, you should use a local separator inside iterator:

```
[<v> (**elems <h> ";")] ;
```

General variables within the iterator are repeated for each element in the list:

```
[<v> (*first <h> ":" <h> **rest)] ;
```

might result in something like:

$$\boxed{\begin{array}{l} first : rest_1 \\ first : rest_2 \\ \ldots \end{array}}$$

Example *Separators vs. Terminators*

The concrete syntax definition of a language may include lists in which elements are either separated or terminated by a token such as a semi-colon. It is important to preserve the intention of the token in the pretty printer as well, so that trees may be edited as expected, based on the concrete syntax definition.

For example, the METAL rules:

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

indicates that the semi-colon serves as a separator of expressions, whereas the rules:

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

imply that the semi-colon is a terminator of expressions.

Respecting the first METAL rule, the semi-colon as separator should be pretty printed with the rule:

```
exp_s(*exp,**exps) → [<hv> *exp (<h> ";" **exps)] ;
```

Note that the semi-colon does not appear when the list contains only one element since `**exps` is empty. The semi-colon as terminator should be pretty printed with the rule:

```
exp_s(**exps) → [<hv> (**exps <h> ";")] ;
```

Notes

- Terminals (commas, semi-colons, etc.) in the iterator structure do not respond to the mouse click.

3.3.3 Control constructs: If and Case

Syntax

if = " `if`" *cond* " `then`" *elem_list*
[*elsif*] [" `else`" *elem_list*]
" `end`" " `if`"
elsif = [*elsif*] " `elsif`" *cond* " `then`"
elem_list

Syntax

case = " `case`" *selector of clause_list*
" `end`" " `case`"
selector = *named_var* | *printlevel_name* | *arith_func_call*
clause_list = {*clause* " `;`"}∗
clause = *choice_list* " `:`" *elem_list*
choice_list = *choice* {" `,`" *choice*}∗
choice = *simple_patt* | *integer* | " `others`"

Description

These two control constructs allow you to group similar rules together and thus significantly reduce the size of a PPML program. Their advantages are best illustrated through examples.

Example *The if-then-else problem*

A pretty printer for a language with an `if` instruction might include the following rules to treat the cases of "if-then" and "if-then-else":

```
#if(*cond, *stat, null_tree) ->
   [<v> [<hv> "if" *cond]
        [<hv> "then" *stat]
        [<h> "end" "if"] ;

#if(*cond, *stat1, *stat2) ->
   [<v> [<hv> "if" *cond]
        [<hv> "then" *stat1]
        [<hv> "else" *stat2]
        [<h> "end" "if"] ;
```

The Case and If constructs allow you to select and pretty print according to specific values of a variable. Thus, the single compact rule accomplishes the same task:

```
#if(*cond, *stat1, *stat2) ->
   [<v> [<hv> "if" *cond]
        [<hv> "then" *stat1]
        case *stat2 of
```

```
            null_tree : ;
            others    : [<hv> "else" *stat2];
         end case
         [<h> "end" "if"] ;
```

Note that instructions within a Case statement must conclude with a semicolon. Since there are only two formatting possibilities in this example, we could have just as easily exploited the If construct:

```
#if(*cond, *stat1, *stat2) ->
   [<v> [<hv> "if" *cond]
        [<hv> "then" *stat1]
        if *stat2 in {null_tree} then
        else [<hv> "else" *stat2]
        end if
        [<h> "end" "if"] ;
```

Please consult the section on "Constrained patterns" for the types of conditional statements valid in the If construct.

Example *Using Case with print level values*

It is possible to format according to the current print level, as shown in the following rule:

```
op(*x) !pl ->
   [<h> case pl of
           0 : "#" ;
           1 : "+" ;
           2 : "*" ;
           others : *x ;
        end case ] ;
```

Example *Printing the phylum name*

You might also find it useful to print phylum name of the current operator when the print level is zero, as in:

```
EXP + EXP
```

instead of simply:

```
# + #
```

To accomplish this, define an external function call that finds the phylum of the current node with VTP functions:

```
(de ppLstd:atm:getphylum (tree)
   (let ((phylum ({phylum}:name
                     ({tree}:son_phylum
                        ({tree}:up tree 1)
                        ({tree}:rank tree)))))
     (string phylum)
   )
)
```

to be called in a Case statement as follows:

```
op(*x) !pl ->
   [<h> case pl of
           0 : getphylum(*x) ;
           others : *x ;
        end case ] ;
```

Example *Using Case inside an iteration*

The following rule is useful if you want to pretty print list terminators according to the list element type:

```
*x where *x in {list1(**y),list2(**y),list3(**y)} ->
   [<hv> (**y <h> case *x of
                        list1 : ";" ;
                        list2 : "." ;
                        list3 : "?" ;
                  end case )] ;
```

Notes

- If *selector* is a *printlevel_name* or *arith_func_call*, the corresponding *choices* must be integers.

3.4 Pretty printer resources

Syntax

class_descr	=	" `in`" (" `class`" | " `term class`" " `:`") *elem_with_attributes*
elem_with_attributes	=	*box* | *terminal* | *ext_func_call*

Description

Any box or terminal string may be surrounded by a *class specification*. The class specification allows you to define font and foreground and background colors for tokens. A class specification **in class =** or **in term class =** that precedes a terminal or external function call affects the terminal or function call only. A class specification **in class =** that precedes a box affects all terminals and function calls as well as recursive calls at run time. A class specification **in term class =** that precedes a box is an abbreviation for putting **in class =** in front of every terminal and every function call in the box. This specification does not affect the formatting of recursive calls. In other words, the right hand side:

```
in term class = foo :
   [<v> "t1" ext_call(*x) "t2" [<h> "t3" *z] *y]
```

is strictly equivalent to:

```
[<v> in class = foo :"t1"
     in class = foo :ext_call(*x)
     in class = foo :"t2"
     [<h> in class = foo :"t3" *z] *y]
```

On the other hand, the specification:

```
in class = foo : [<v> "t1" ext_call(*x) "t2" [<h> "t3" *z] *y]
```

affects all the terminals and function calls on the right hand side *and* all terminals formatted during the recursive call to `*x`, `*y`, and `*z`.

Class specifications may be inherited from surrounding rules. Inheritance works as follows:

- Specifying a class overrides inherited classes.
- A class is otherwise inherited by the parent box, or its parent box, etc., up to the root.

One may then specify resources for these classes. Please consult the resource manual for the semantics of resource inheritance.

Example

The following rule specifies a class for the `begin` and `end` tokens of a fixed arity operator **op**:

```
op(*x,*y) -> [<v> in class = opclass : "begin"
                  *x *y
                  in class = opclass : "end"];
```

If the specification precedes the box, the font and color resources apply to the tokens formatted by the recursive calls to `*x` and `*y` as well as `begin` and `end`:

```
op(*x,*y) -> [<v> in class = opclass : [<v> "begin" *x *y "end"]];
```

Notes

In the current version of PPML:

- a class specification may not precede a "bare" recursive call. For example, you may not write `in class = special : *x`.
- the right side of a rule must be a box. You may not write:

```
op(*x) -> in class = myclass : [<h> "text" *x];
```

but you may write:

```
op(*x) -> [<h> in class = myclass : [<h> "text" *x];
```

3.5 Contexts

Syntax

context_patt = *context_name* " ::"
(simple_patt | *constraint_patt*
| *printlevel_patt* | *annotated_patt)*
context_name = *identifier* | *integer*

Description

Occasionally, abstract syntax operators need *contextual* pretty printing. A context is another set of PPML rules for the same language, located in the same pretty printer. We pass into a context with a recursive call prefaced by the context name. This restricts PPML's vision to rules whose patterns are also prefaced by the context name. Patterns not defined in a named context are said to belong to the context zero. If no rule applies in a given context, the formatting machine invokes the generic pretty printer.

Example *The "dangling else"*

The classic example illustrating the purpose of contexts involves the "dangling else" of the Pascal `if` statement. In Pascal, any `if` statement found in the `then` part of another `if` statement must also include an `else` part and possibly other instructions. Thus, the `if` rules described in the section on Case and If constructs:

```
#if(*cond, *stat, null_tree) ->
   [<v> [<hv> "if" *cond]
        [<hv> "then" *stat]
        [<h> "end" "if"] ;

#if(*cond, *stat1, *stat2) ->
   [<v> [<hv> "if" *cond]
        [<hv> "then" *stat1]
        [<hv> "else" *stat2]
        [<h> "end" "if"] ;
```

are not sufficient for pretty printing Pascal since they don't take into account the necessity of an `else` part.

The problem "when we are in the `then` part of an `if` statement?" suggests that we should enter a new context any time we enter the `then` part of an `if` statement that also has an `else` part. The solution, exploiting the context "dangling," requires three rules:

```
#if(*cond, *stat, null_tree) ->
   [<v> [<hv> "if" *cond]
        [<hv> "then" *stat]
        [<h> "end" "if"] ;

#if(*cond, *stat1, *stat2) ->
   [<v> [<hv> "if" *cond]
        [<hv> "then" dangling::*stat1]
        [<hv> "else" *stat2]
        [<h> "end" "if"] ;

dangling::#if(*cond, *stat1, *stat2) ->
   [<v> [<hv> "if" *cond]
        [<hv> "then" dangling::*stat1]
        [<hv> "else" dangling::*stat2]
        [<h> "end" "if"] ;
```

The first rule pretty prints isolated `if` statements without an `else` part. The second rule pretty prints `if` statements with an `else` part and introduces the "dangling" context. Once in the dangling context, any future `if` statements remain in the dangling context, thus ensuring that each as an `else` part.

Remember that if no rule in a given context applies to a given node, the formatting machine looks in the context zero. It is thus unnecessary to include the same rule in a special context and the context zero.

3.6 Initializations

Syntax

init_part = [*constant_part*] [*default_part*]

Description

The optional initialization section of a PPML program allows you to define constants and default values.

3.6.1 Constants

Syntax

constant_part	=	" `constant`" *const_decl_list*
const_decl_list	=	{*const_decl*}∗
const_decl	=	*identifier* " `=`" *integer* " `;`"

Description

In this optional section, you may define integer constants to be used as separator parameters. Constants may be assigned different values in different chapters; their visiblity is governed by the usual rules of block structures.

Example

```
constant
      tab = 5 ;
      smalltab = 2 ;
```

may be used as follows:

```
op(*x,*y) → [<hv smalltab,tab,0> *x ";" <h tab,0> *y] ;
```

3.6.2 Defaults

Syntax

default_part	=	" `default`" *def_decl_list*
def_decl_list	=	{*default_decl* " `;`"}∗
default_decl	=	*default_sep* | *attribute*
attribute	=	*identifier* " `=`" *attribute_value*
attribute_value	=	*integer* | *identifier*

Description

In this optional section, you may define default values for separators. If you specify default values for a separator, you must specify values for

each parameter of the separator. Parameters of unspecified separators are initialized to zero. You may use constants defined in the "constant" section as parameter values:

Example

```
default
      <h 1> ; <v smalltab,1> ;
```

The usual rules of block structures govern the visiblity of default values.

It is also possible, during the course of pretty printing, to encounter abstract syntax trees from other languages. This may occur either for atomic operators implemented as TREE or annotations whose values are trees. In the *first* default section of a PPML program, you may specify pretty printers and starting contexts for pretty printing "foreign" trees.

Example

```
default
      <h 1> ;
      METAL-pprinter = special ;
      METAL-context = aux ;
      PPML-context = aux1 ;
      pprinter = std1 ;
```

In this example, any METAL tree will be formatted with the pretty printer `special` starting in the context `aux`. Any PPML tree will be pretty printed with `std1` (since PPML-pprinter is not mentioned and the default pretty printer `pprinter` is `std1`) starting in the context `aux1`. Finally, any other tree will be pretty printed with `std1` starting in the *current* context of the original pretty printer, be it zero or some special context.

Notes

- If the formatting machine looks for a context not defined in the foreign pretty printer, an error is returned and pretty printing fails.
- The formatting machine looks for foreign language pretty printer name in the following order:
  - a name specified in the default section (eg. `METAL-pprinter = std;`).

– the default pretty printer name for foreign languages (`pprinter = std1;`).

– a pretty printer with the same name as the current pretty printer

The formatting machine then loads the pretty printer with the selected name. If it doesn't exist, the generic pretty printer is invoked.

- It is possible to specify a default starting context instead of using the current context. For example, to begin systematically in the zero context, include the instruction:

```
context = 0 ;
```

3.7 Zones

Syntax

zone_list = {*zone*}∗
zone = *phyla_part* | *function_def*
| *chapter* | *rule_part*

Zones may appear several times and in any order in a PPML program.

3.7.1 Phyla declarations

Syntax

phyla_part = " `phyla`" *phyla_decl_list*
phyla_decl_list = {*phylum_decl*}∗
phylum_decl = *phylum_name* " `=`" *phylum_spec* " `;`"
phylum_spec = *phylum_name*
| " {" *pattern_list* " }"
| *phylum_spec* (" `+`" | " `-`") *phylum_spec*
phylum_name = *identifier*

Description

Phyla are groups of possible descendant operators for a given operator. Although phyla definitions normally appear in the abstract syntax definition for the object language, it is possible to define additional phyla (called

"local" phyla) in a PPML program for pattern matching purposes. Abstract syntax, local phyla, and pattern lists appear in conjunction with the *union* and *difference* operators in the constrained patterns and control structures discussed earlier.

Example

```
phyla
     TERMINALS = ident,integer,string ;
     OTHERS = EVERY - TERMINALS ;
```

Notes

The phylum EVERY is a METAL predefined phylum that includes all operators of the object language.

3.7.2 Function declarations

Syntax

function_def	=	" `function`" *identifier* " `is`" *func_clause_list* " `end`" *identifier* " `;`"
func_clause_list	=	{*func_clause* " `;`"}+
func_clause	=	*identifier* " `(`" (*simple_patt* | *constraint_patt*) " `)`" " `=`" *arith_exp*
arith_exp	=	*integer* | *identifier* | *function_call*
function_call	=	*identifier* " `(`" *named_var* " `)`"

Description

A PPML function associates a pattern with an integer for use in constrained patterns and control structures. During execution of a function, the formatting machine evaluates each clause in succession until a match is found, and returns the integer corresponding to the matched pattern. If no pattern matches, pretty printing fails.

Example

```
function func is
  func(op1) = 1 ;
  func(*x where *x in {op2,op3}) = 2 ;
  func(*x where *x in TERMINALS) = 3 ;
  func(op4(*x)) = func(*x) ;
  func(*x) = 4 ;
end func ;
```

Notes

- Note that an arithmetic expression may be a function call.
- You may use a "catch all" rule as shown to prevent the pretty printer from failing when it can't match a function pattern, but this is dangerous as it hides bugs.

Example *Operator precedence*

```
function prec is
  prec(*a where *a in {eql,neq,leq,#geq,gtr,#in}) = 4 ;
  prec(*a where *a in {plus,minus,#or}) = 3 ;
  prec(*a where *a in {uplus,uminus}) = 2 ;
  prec(*a where *a in {mult,div,intdiv,mod,#and}) = 1 ;
  prec(*a) = 0 ;
end prec ;
```

This function illustrates one common application of PPML functions concerns the precedence of operators of a language. The concrete syntax of a language often includes structures such as parentheses or blocks that prioritize otherwise ambiguous operators. Although the hierarchy of the abstract syntax tree reflects the influence of priority constructs, the tokens themselves are lost It's the pretty printer's task to reproduce this priority information where necessary. By using functions, you may associate integers with operators and use control structures to reproduce parentheses, etc. when appropriate.[17]

As was the case for atomic pretty printing, it is possible to define more complicated mapping functions externally. These functions should be stored in a file called:

[17]Please consult "A Centaur Tutorial" for the example of parentheses.

`language-pp.at`

where `language` is the name of the object language and `pp` is the name of a pretty printer for `language`.

These functions must obey the following Lisp package naming scheme:

`#:pp`*language*`prettyprinter:int:myfunction`

where *language* is the name of the object language, **prettyprinter** is the name of the pertinent pretty printer, and `myfunction` is the desired name of the function.

These functions must take one argument (which may be anything) must return an integer.

For example, suppose we wanted to define the function `prec` for the `std` pretty printer of the language `pascal`. In the `pascal-std.at` we would include the function:

```
(de #:pppascalstd:int:prec (tree)
   (let ((op ({operator}:name ({tree}:operator tree))))
     (selectq op
        ((eql neq leq geq gtr in) 4)
        ((plus minus or) 3)
        ((uplus uminus) 2)
        ((mult div intdiv mod and) 1)
        (t 0)
     )
   )
)
```

which uses VTP primitives to extract the operator name of the `tree` argument.

3.8 Chapters

Syntax

chapter	=	" `chapter`" *identifier* [*context_part*] [*init_part*] *zone_list* " `end`" " `chapter`" " `;`"
context_part	=	" `with`" *identifier*

Description

Chapters enable provide a certain structure a PPML program. They are useful not only for organizational purposes, but to allow you to redefine default and constant values within the chapter. Specifying a context for the chapter causes all rules to be defined (unless explicity overridden), and all recursive calls to occur in that context.

Example

```
chapter PARENTHESES with paren
default
  <h 0> ;
rules
  *x where *x in EXP → [<h> "(" 0::*x ")" ] ;
  *x → [<h> "((" 0::*x "))" ] ;
end chapter ;
```

In this example, the two rules indicated exist in the context `paren`, but the recursive calls on `*x` refer to the 0 context. The above chapter is equivalent to:[18]

Example

```
chapter PARENTHESES
default
  <h 0> ;
rules
  paren::*x where *x in EXP → [<h> "(" *x ")" ] ;
  paren::*x → [<h> "((" *x "))" ] ;
end chapter ;
```

3.9 Programs

Syntax

[18] Assuming this chapter isn't defined within another chapter, in which case, the default context may not necessarily be the 0 context.

program = “ `prettyprinter`” [*pp_name*] “ `of`” *language_name*
“ `is`” [*init_part*] *zone_list* “ `end`” “ `prettyprinter`”
pp_name = *identifier* | *integer*
language_name = *identifier*

Description

The *pp_name* argument allows you to define several pretty printers for the language *language_name*.

Notes

Nameless programs are called `std` by default. This name appears in the Centaur editor when the Ppml program is read.

4 Ppml errors and warnings

The following sections contain examples of Ppml environment errors and Ppml compiler errors. In each example, error messages appear at the bottom of the error box in *italics* and the location of the erroneous code in **boldface**. Only errors prevent the Ppml program from being compiled.

4.1 General errors

4.1.1 Toplevel errors

Occasionally, the formatting machine fails, generating a Lisp error and the message:

```
** vref :  argument hors limite :  3
```

Two problems may produce this error:

- The tables for the language and the compiled Ppml code do not not correspond, so you must recompile the Ppml program.
- You have tried to make a recursive call on a value that is not a tree, for example the value of an atom or a frame that is a string, integer, etc. You must treat these values with either predefined or externally defined functions, as demonstrated in the reference manual.

4.1.2 Illegal argument to compiler

This error indicates that the function `{ppml}:compile` has been called with an illegal argument. Please consult the section on Ppml primitives for information about Ppml compiler functions.

4.2 Compiler errors

The following section contain examples of Ppml compiler errors and error messages.

4.2.1 Unreachable code eliminated

rules

*x → [<h> "[" *x "]"];
op(*x) → [<**h**> "(" *x ")"] ;

Warning: Unreachable code has been eliminated

General patterns may not precede more specific patterns for the same operator(s). In this example, the pattern *x occludes all succeeding rules.

4.2.2 Undefined default

default

<v 0,3>; <hv 1,0,0>;
rules

op(*x) → [<**mysep**> "[" *x "]"] ;

Undefined default for the mysep separator

Excluding the predefined operators <h>, <v>, <v1>, <hv>, and <hov>, it is only legal to refer to a separator without parameters if default values have been specified in a default section of the program. Thus, even though the separator <mysep> doesn't exist, the following program excerpt will not provoke an error:

default

<v 0,3>; <mysep 1,0,0>;
rules

op(*x) → [<mysep> "[" *x "]"] ;

The compiler accepts undefined separator names for two reasons:

- so that truly motivated CENTAUR hackers can define new separators,
- PPML doesn't yet have an adequate type checker.

4.2.3 Undefined constant

```
constant

   tab = 3 ;
rules

   op(*x) → [<h mytab> "[" *x "]" ] ;

The constant mytab is not defined
```

All constants (such as `mytab`) must be defined in a constant declaration section.

4.2.4 Undefined operator

```
rules

   unxpctd(*x) → [<h> "[" *x "]" ] ;

Undefined operator unxpctd
```

Operators referenced in a PPML program must belong to the abstract syntax of the object language. Thus, we gather from this example that either `unxpctd` does not exist at all in the abstract syntax definition, or a valid operator name has been misspelled.

4.2.5 Undefined phylum

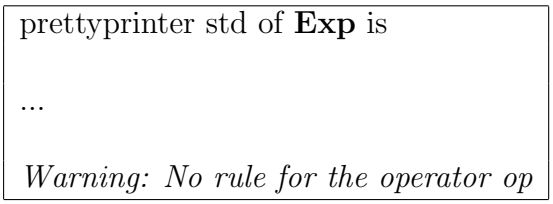
```
rules

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

Undefined phylum EXP
```

Phyla referenced in a Ppml program must either belong to the abstract syntax of the object language or be defined in the Ppml program itself. We deduce from this example that `EXP` does not exist in either.

4.2.6 No rule for an operator

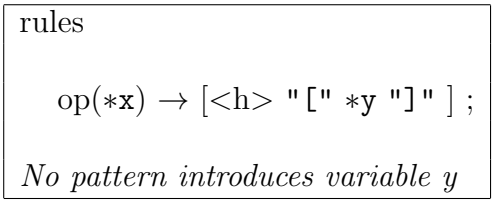


The compiler warns that no pattern can ever match a given operator (here `op`) by highlighting the name of the language. Note that a general pattern such as:

```
*x !0 → ...
```

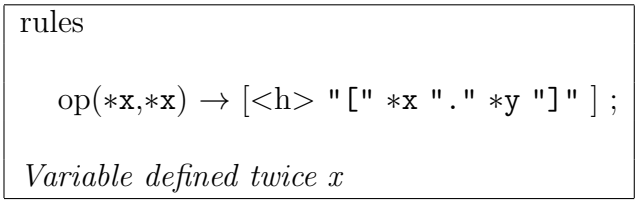
applies to any operator and thus eliminates the warning message.

4.2.7 Unspecified variable



Variables that appear on the right hand side of a rule must also appear on the left hand side. On the other hand, it is possible, though wasteful, to ignore on the right side variables that appear on the left.

4.2.8 Double declaration of variable



You may only use a variable name once in a pattern.

4.2.9 Illegal use of variable

```
rules

   op(*x,**oplist) → [<h> "[" *x "." (**oplist) "]" ] ;

Illegal use of the variable oplist
```

In this example, `op` is a fixed arity operator with two descendents, so it is illegal to use a list variable anywhere in the pattern. The rule:

```
rules

   op(**extralist,**oplist) → [<h> "[" *x "." (**oplist) "]" ] ;

Illegal use of the variable oplist
```

also provokes this error since only one list variable may appear in a pattern. Finally the rule:

```
rules

   op(*x,**oplist) → [<h> "[" *x "." **oplist "]" ] ;

Illegal use of the variable oplist
```

provokes this error since the list variable in the formatting part is not in the iterator construct.

Notes

Recall that you may use as many general variables as you like in the pattern of a list operator, implying that the pattern corresponds to a precise number of elements in the list.

4.2.10 Pattern may be simplified

```
rules

   *x where *x in {op1(*, *), op2(*, *) →
      [<h> getsymbol(*x) ] ;

Warning: Pattern may be simplified op1
Warning: Pattern may be simplified op2
```

It is possible to simplify patterns for fixed arity operators whose descendents are all represented by anonymous variables. The simplified pattern for this example is:

```
*x where *x in {op1, op2} →
      [<h> getsymbol(*x) ] ;
```

4.2.11 Illegal pattern

```
function prec is
   prec(op(*x) where *x in EXP) = 4 ;     ...
end prec ;
Illegal pattern
```

Nested patterns are not permitted within function definitions.

4.2.12 Illegal separator list

```
rules

   op(*x) → [<h tab1,tab2> "!" *x ] ;

Illegal list of parameters in h separator
```

Separators have a fixed number of arguments (the <h> separator has one argument). You must either use the default form of a separator (no arguments explicit) or its complete form (all arguments explicit, the exact number of arguments).

4.2.13 Wrong type

```
function prec is
    prec(*op where op in EXP) = 4 ;
    ...
end prec ;

Wrong type for op
```

The syntax of PPML allows the keyword `where` to be followed by any simple pattern, making the above function excerpt parsable, but semantically incorrect. The compiler demands that the pattern following the word `where` be a variable.

4.2.14 Wrong kind of list

```
rules

    op_s() → [<h> "token"] ;

At least one son is needed for list operator op_s
```

Patterns for list operators such as `op_s` that are defined as "plus" lists must allow for at least one list element. To match the node `op_s` without regard for the descendent list, use the pattern:

```
op_s → [<h> "token"] ;
```

4.2.15 Wrong arity for operator

```
rules

    plus(*exp1, *exp2, *exp3) →
        [<h> *exp1 ":" *exp2 "," *exp3] ;

The operator plus is used with a wrong arity
```

A pattern for a fixed arity operator much exactly match the arity of the operator. In this example, we suppose that `plus` is a binary operator, so it is illegal to specify three descendents.

4.2.16 Operator is atomic

rules

ident(∗exp1) → [<h> "!" ∗exp1] ;

The operator ident is atomic

You may not match descendents of an atomic operator. Remember to use the syntax:

```
ident ∗exp1 → [<h> "!" ∗exp1] ;
```

to pretty print the value of an atomic operator, here stored in `∗exp1`.

4.2.17 Operator is not atomic

rules

plus ∗x → [<h> ∗x] ;

The operator is not atomic

Non-atomic trees do not have values associated, so it is illegal to match for values.

4.2.18 Operator is not implemented as string

rules

integer 'foo' → [<h>];

The operator integer is not implemented as a string.

You may not write patterns to match the value of an atomic operator not implemented as a string. In this example, the operator **integer** has been implemented as `INTEGER`, for example.

4.2.19 Iterator requires list variable

rules

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

No sublist meta-variable in iterator construction

The iterator must contain one (and only one) list variable. Note that you may include as many general variables, tokens, and separators as you wish, as long as there is one list variable.

5 The syntax of Ppml

alphanum = *wordsep* (*letter* | *digit*)
annotated_patt = (*simple_patt* | *constraint_patt* | *printlevel_patt*)
frame_list
anonymous_var = " *"
arg = *named_var* | *string* | *frame*
arith_exp = *integer* | *identifier* | *function_call*
atomarg = *variable* | *string*
attribute_value = *integer* | *identifier*
attribute = *identifier* " =" *attribute_value*
bool_exp = *arith_exp*
| *arith_exp* (" =" | " >" | " >=" | " <" | " <=")
" not" " (" *bool_exp* ")"
| *bool_exp* (" and" | " or") *bool_exp*
| *var_list* " in" *phylum_spec*
box = " [" *global_sep elem_list* "]"
case = " case" *selector of clause_list*
" end" " case"
class_descr = " in" (" class" | " term class") " =" *identifier* " :"
elem_with_attributes
elem_with_attributes = *box* | *terminal* | *ext_func_call*
chapter = " chapter" *identifier* [*context_part*] [*init_part*]
zone_list " end" " chapter" " ;"
choice_list = *choice* {" ," *choice*}∗
choice = *simple_patt* | *integer* | " others"
clause_list = {*clause* " ;"}∗
clause = *choice_list* " :" *elem_list*
cond_list = *cond* {" ," *cond*}∗
cond = *bool_exp*
const_decl_list = {*const_decl*}∗
const_decl = *identifier* " =" *integer* " ;"
constant_part = " constant" *const_decl_list*
constraint_patt = *simple_patt* " where" *cond_list*

context_name = *identifier* | *integer*
context_part = “ `with`” *identifier*
context_patt = *context_name* “ `::`”
(simple_patt | *constraint_patt*
| *printlevel_patt* | *annotated_patt)*
def_decl_list = {*default_decl* “ `;`”}∗
default_decl = *default_sep* | *attribute*
default_part = “ `default`” *def_decl_list*
default_sep = “ <” *type_name param_list* “ >”
digit = *0* | *1* | ... *9*
elem_list = {*elem*}∗
elem = *terminal* | *recursive_call* | *box*
| *if* | *case* | *sublist_iterator*
| *ext_func_call* | *local_sep*
elsif = [*elsif*] “ `elsif`” *cond* “ `then`”
elem_list
ext_func_call = *ext_func_name* “ `(`” *arg* “ `)`”
ext_func_name = *identifier*
formatting_part = *box*
frame_list = {*frame*}∗
frame_name = *identifier*
frame = “ ↑” *frame_name*
func_clause_list = {*func_clause* “ `;`”}+
func_clause = *identifier* “ `(`” (*simple_patt* | *constraint_patt*) “ `)`”
“ `=`” *arith_exp*
function_call = *identifier* “ `(`” *named_var* “ `)`”
function_def = “ `function`” *identifier* “ `is`”
func_clause_list “ `end`” *identifier* “ `;`”
general_var = “ ∗” *var_name*
global_sep = *separator*
identifier = [“ `#`”] *letter* {*alphanum*}∗
if = “ `if`” *cond* “ `then`” *elem_list*
[*elsif*] [“ `else`” *elem_list*]
“ `end`” “ `if`”

init_part	=	[*constant_part*] [*default_part*]
integer	=	{*digit*}+
language_name	=	*identifier*
letter	=	*uppercase* | *lowercase*
list_var	=	" **" *var_name*
local_sep	=	*separator*
lowercase	=	*a* | *b* | *c* | ... | *z*
named_var	=	*general_var* | *list_var*
op_name	=	*identifier*
param_list	=	*param* {" ," *param*}*
param	=	*integer* | *identifier*
pattern_list	=	*simple_pattern* {" ," *simple_pattern*}*
pattern	=	*simple_patt* | *constraint_patt* | *context_patt*
		| *annotated_patt* | *printlevel_patt*
phyla_decl_list	=	{*phylum_decl*}*
phyla_part	=	" `phyla`" *phyla_decl_list*
phylum_decl	=	*phylum_name* " =" *phylum_spec* " ;"
phylum_name	=	*identifier*
phylum_spec	=	*phylum_name*
		| " {" *pattern_list* " }"
		| *phylum_spec* (" +" | " -") *phylum_spec*
pp_name	=	*identifier* | *integer*
printlevel_exp	=	[*printlevel_name*] (" +" | " -") *integer*
		| *printlevel_name* | *integer*
printlevel_name	=	*identifier*
printlevel_patt	=	(*simple_patt* | *constraint_patt*) " !" *printlevel*
printlevel	=	*printlevel_name* | *integer*
program	=	" `prettyprinter`" [*pp_name*]
		" `of`" *language_name*
		" `is`" [*init_part*] *zone_list*
		" `end`" " `prettyprinter`"
quote_image	=	" ''"
recursive_call	=	[*context_name* " ::"] *simple_call*
		[" !" *printlevel_exp*]
repeated_quote_image	=	" """

rule_list	=	{*rule*}+
rule_part	=	" `rules`" *rule_list*
rule	=	*pattern* " →" *formatting_part* " `;`"
selector	=	*named_var* | *printlevel_name* | *arith_func_call*
separator	=	" <" *type_name* [*param_list*] " >"
simple_call	=	*variable* | *frame*
simple_patt	=	*variable*
		| *op_name* [" `(`" [*pattern_list*] " `)`"]
		| *op_name atomarg* | *op_name*
string_character	=	*any_character_except_CR_or_quote*
string_elem	=	*quote_image* | *string_character*
string	=	" `'`"{*string_elem*}∗" `'`"
sublist_operator	=	[" \"]" `(`" *elem_list* " `)`"
terminal_character	=	*any_character_except_CR_or_repeated_quote*
terminal_elem	=	*repeated_quote_image* | *terminal_character*
terminal	=	" `"`" {*terminal_elem*}∗" `"`"
uppercase	=	*A* | *B* | *C* | ... | *Z*
var_list	=	*named_var* {" `,`" *named_var*}∗
var_name	=	*identifier*
variable	=	*anonymous_var* | *named_var*
wordsep	=	" `_`" | " `-`"
zone_list	=	{*zone*}∗
zone	=	*phyla_part* | *function_def*
		| *chapter* | *rule_part*

6 The Metal definition of Ppml

```
definition of PPML is
   rules
      <AXIOME> ::= <PPRINTER> ;
         <PPRINTER>
      <AXIOME> ::= <PHYLUM> ;
         <PHYLUM>
      <PPRINTER> ::=
         "prettyprinter" <PPRINTER_NAME> "of"
            <IDENT> "is" <INIT> <ZONE_S>
            "end" "prettyprinter" ;
         pprinter (<PPRINTER_NAME>, <IDENT>, <INIT>, <ZONE_S>)
      <PHYLUM> ::= "[PPRINTER]" <PPRINTER> ;
         <PPRINTER>
      <PPRINTER_NAME> ::= ;
         ident-atom ('std')
      <PPRINTER_NAME> ::= <ID_INTEGER> ;
         <ID_INTEGER>
      <PHYLUM> ::= "[PPRINTER_NAME]" <PPRINTER_NAME> ;
         <PPRINTER_NAME>

   abstract syntax
      pprinter  -> PPRINTER_NAME IDENT INIT ZONE_S ;
      PPRINTER  ::= pprinter ;
      PPRINTER_NAME  ::= ID_INTEGER ;
```

```
chapter INIT
   rules
      <INIT> ::= <CONSTANT_OPTION> <DEFAULT_OPTION> ;
         init (<CONSTANT_OPTION>, <DEFAULT_OPTION>)
      <PHYLUM> ::= "[INIT]" <INIT> ;
         <INIT>
      <CONSTANT_OPTION> ::= "constant" <CONST_DECL_S> ;
         <CONST_DECL_S>
      <CONSTANT_OPTION> ::= ;
         constant-list (())
      <PHYLUM> ::= "[CONSTANT]" <CONSTANT_OPTION> ;
         <CONSTANT_OPTION>
      <CONST_DECL_S> ::= <CONST_DECL> ;
         constant-list ((<CONST_DECL>))
      <CONST_DECL_S> ::= <CONST_DECL_S> <CONST_DECL> ;
         constant-post (<CONST_DECL_S>, <CONST_DECL>)
      <PHYLUM> ::= "[CONSTANT]" <CONST_DECL_S> ;
         <CONST_DECL_S>
      <CONST_DECL> ::= <IDENT> "=" <INT_CONST> ";" ;
         const_decl (<IDENT>, <INT_CONST>)
      <PHYLUM> ::= "[CONST_DECL]" <CONST_DECL> ;
         <CONST_DECL>
      <DEFAULT_OPTION> ::= "default" <DEFAULT_DECL_S> ;
         <DEFAULT_DECL_S>
      <DEFAULT_OPTION> ::= ;
         default-list (())
      <PHYLUM> ::= "[DEFAULT]" <DEFAULT_OPTION> ;
         <DEFAULT_OPTION>
      <DEFAULT_DECL_S> ::= <DEFAULT_DECL> ;
         default-list ((<DEFAULT_DECL>))
      <DEFAULT_DECL_S> ::= <DEFAULT_DECL_S> <DEFAULT_DECL> ;
         default-post (<DEFAULT_DECL_S>, <DEFAULT_DECL>)
```

```
      <PHYLUM> ::= "[DEFAULT]" <DEFAULT_DECL_S> ;
         <DEFAULT_DECL_S>
      <DEFAULT_DECL> ::= <DEFAULT_SPEC> ";" ;
         <DEFAULT_SPEC>
      <PHYLUM> ::= "[DEFAULT_DECL]" <DEFAULT_SPEC> ;
         <DEFAULT_SPEC>
      <DEFAULT_SPEC> ::= <SEP> ;
         <SEP>
      <DEFAULT_SPEC> ::= <ATTRIBUT> ;
         <ATTRIBUT>
      <DEFAULT_SPEC> ::= <FONT_DESC> ;
         <FONT_DESC>

   abstract syntax
      init  -> CONSTANT DEFAULT ;
      constant  -> CONST_DECL * ... ;
      const_decl  -> IDENT INT_CONST ;
      default  -> DEFAULT_DECL * ... ;
      INIT  ::= init ;
      CONSTANT  ::= constant ;
      CONST_DECL  ::= const_decl ;
      DEFAULT  ::= default ;
      DEFAULT_DECL  ::= sep attribut font_desc ;
end chapter ;
```

```
chapter FONT_DESCRIPTION
   rules
      <FONT_DESC> ::= <IDENT> "(" <FONT_ATTRIB_S> ")" ;
         font_desc (<IDENT>, <FONT_ATTRIB_S>)
      <FONT_ATTRIB_S> ::= ;
         font_attrib_s-list (())
      <FONT_ATTRIB_S> ::= <FONT_ATTRIB_S> <FONT_ATTRIB> ;
         font_attrib_s-post (<FONT_ATTRIB_S>, <FONT_ATTRIB>)
      <PHYLUM> ::= "[FONT_ATTRIB_S]" <FONT_ATTRIB_S> ;
         <FONT_ATTRIB_S>
      <FONT_ATTRIB> ::= <IDENT> ":" <ID_INTEGER> ;
         font_attrib (<IDENT>, <ID_INTEGER>)
      <PHYLUM> ::= "[FONT_ATTRIB]" <FONT_ATTRIB> ;
         <FONT_ATTRIB>

   abstract syntax
      font_desc  -> IDENT FONT_ATTRIB_S ;
      font_attrib_s  -> FONT_ATTRIB * ... ;
      font_attrib  -> IDENT ID_INTEGER ;
      FONT_ATTRIB_S  ::= font_attrib_s ;
      FONT_ATTRIB  ::= font_attrib ;
end chapter ;
```

```
chapter ZONE_S
   rules
      <ZONE_S> ::= ;
         zone_s-list (())
      <ZONE_S> ::= <ZONE_S> <ZONE> ;
         zone_s-post (<ZONE_S>, <ZONE>)
      <PHYLUM> ::= "[ZONE_S]" <ZONE_S> ;
         <ZONE_S>
      <ZONE> ::= <RULE_PART> ;
         <RULE_PART>
      <ZONE> ::= <SORT_PART> ;
         <SORT_PART>
      <ZONE> ::= <FUNC_DEF> ;
         <FUNC_DEF>
      <ZONE> ::= <CHAPT> ;
         <CHAPT>
      <PHYLUM> ::= "[ZONE]" <ZONE> ;
         <ZONE>

   abstract syntax
      zone_s  -> ZONE * ... ;
      ZONE_S  ::= zone_s ;
      ZONE  ::= rule_s sort_def_s func_def chapt ;
end chapter ;
```

```
chapter RULE
   rules
      <RULE_PART> ::= "rules" <RULE_S> ;
         <RULE_S>
      <RULE_S> ::= <RULE> ;
         rule_s-list ((<RULE>))
      <RULE_S> ::= <RULE_S> <RULE> ;
         rule_s-post (<RULE_S>, <RULE>)
      <RULE> ::= <CFHC_PATT> "->" <RIGHT_SIDE> ";" ;
         rule (<CFHC_PATT>, <RIGHT_SIDE>)
      <PHYLUM> ::= "[RULE]" <RULE> ;
         <RULE>
      <RIGHT_SIDE> ::= ;
         box
            (sep (ident-atom ('v1'), s_param_s-list ()),
                elem_s-list ())
      <RIGHT_SIDE> ::= <BOX> ;
         <BOX>

   abstract syntax
      rule_s  -> RULE + ... ;
      rule  -> CFHC_PATT BOX ;
      RULE  ::= rule ;
end chapter ;
```

```
chapter SORT_DEF
   rules
      <SORT_PART> ::= "phyla" <SORT_DEF_S> ;
         <SORT_DEF_S>
      <SORT_DEF_S> ::= <SORT_DEF> ;
         sort_def_s-list ((<SORT_DEF>))
      <SORT_DEF_S> ::= <SORT_DEF_S> <SORT_DEF> ;
         sort_def_s-post (<SORT_DEF_S>, <SORT_DEF>)
      <SORT_DEF> ::= <IDENT> "=" <SORT_S> ";" ;
         sort_def (<IDENT>, <SORT_S>)
      <PHYLUM> ::= "[SORT_DEF]" <SORT_DEF> ;
         <SORT_DEF>
      <SORT_S> ::= <SORT> ;
         sort_s-list ((<SORT>))
      <SORT_S> ::= <SORT_S> <SIGNSORT> ;
         sort_s-post (<SORT_S>, <SIGNSORT>)
      <PHYLUM> ::= "[SORT_S]" <SORT_S> ;
         <SORT_S>
      <SIGNSORT> ::= <SORT> ;
         <SORT>
      <SIGNSORT> ::= "+" <SORT> ;
         <SORT>
      <SIGNSORT> ::= "-" <SORT> ;
         minussort (<SORT>)
```

```
      <PHYLUM> ::= "[SIGNSORT]" <SIGNSORT> ;
         <SIGNSORT>
      <SORT> ::= "{" <PATT_S1> "}" ;
         <PATT_S1>
      <SORT> ::= <IDENT> ;
         <IDENT>
      <PHYLUM> ::= "[SORT]" <SORT> ;
         <SORT>
      <PATT_S1> ::= <PATT> ;
         patt_s1-list ((<PATT>))
      <PATT_S1> ::= <PATT_S1> "," <PATT> ;
         patt_s1-post (<PATT_S1>, <PATT>)

   abstract syntax
      sort_def  -> IDENT SORT_S ;
      sort_def_s  -> SORT_DEF + ... ;
      minussort  -> SORT ;
      sort_s  -> SIGNSORT * ... ;
      patt_s1  -> PATT + ... ;
      SORT_DEF  ::= sort_def ;
      SORT_S  ::= sort_s ;
      SIGNSORT  ::= SORT minussort ;
      SORT  ::= patt_s1 ident ;
end chapter ;
```

```
chapter ARITH_FUNCTION_DEF
   rules
      <FUNC_DEF> ::=
         "function" <IDENT> "is" <FUNC_VALUE_S>
                              "end" <IDENT> ";" ;
         func_def (<IDENT>, <FUNC_VALUE_S>)
      <FUNC_VALUE_S> ::= <FUNC_VALUE> ";" ;
         func_value_s-list ((<FUNC_VALUE>))
      <FUNC_VALUE_S> ::= <FUNC_VALUE_S> <FUNC_VALUE> ";" ;
         func_value_s-post (<FUNC_VALUE_S>, <FUNC_VALUE>)
      <PHYLUM> ::= "[FUNC_VALUE_S]" <FUNC_VALUE_S> ;
         <FUNC_VALUE_S>
      <FUNC_VALUE> ::=
         <IDENT> "(" <C_PATT> ")" "=" <ARITH_EXP> ;
         func_value (<IDENT>, <C_PATT>, <ARITH_EXP>)
      <PHYLUM> ::= "[FUNC_VALUE]" <FUNC_VALUE> ;
         <FUNC_VALUE>
      <ARITH_EXP> ::= <ARITH_FUNCTION_CALL> ;
         <ARITH_FUNCTION_CALL>
      <ARITH_EXP> ::= <ID_INTEGER> ;
         <ID_INTEGER>
      <PHYLUM> ::= "[ARITH_EXP]" <ARITH_EXP> ;
         <ARITH_EXP>

   abstract syntax
      func_def  -> IDENT FUNC_VALUE_S ;
      func_value_s  -> FUNC_VALUE + ... ;
      func_value  -> IDENT C_PATT ARITH_EXP ;
      FUNC_VALUE_S  ::= func_value_s ;
      FUNC_VALUE  ::= func_value ;
      ARITH_EXP  ::= ID_INTEGER arith_function_call ;
end chapter ;
```

```
chapter ARITH_FUNCTION_CALL
   rules
      <ARITH_FUNCTION_CALL> ::= <IDENT>
                               "(" <ARITH_FUNCTION_ARG> ")" ;
         arith_function_call (<IDENT>, <ARITH_FUNCTION_ARG>)

   abstract syntax
      arith_function_call  -> IDENT ARITH_FUNCTION_ARG ;
end chapter ;

chapter CHAPT
   rules
      <CHAPT> ::=
         "chapter" <CHAPTER_NAME> <CONTEXT_WITH_OPTION>
            <INIT> <ZONE_S>
            "end" "chapter" ";" ;
         chapt
         (<CHAPTER_NAME>, <CONTEXT_WITH_OPTION>,
          <INIT>, <ZONE_S>)
      <CHAPTER_NAME> ::= <IDENT> ;
          <IDENT>
      <CHAPTER_NAME> ::= <STRING> ;
          <STRING>
      <CONTEXT_WITH_OPTION> ::= "with" <CONTEXT> ;
         <CONTEXT>
      <CONTEXT_WITH_OPTION> ::= ;
         void ()
      <PHYLUM> ::=
         "[CONTEXT_WITH_OPTION]" <CONTEXT_WITH_OPTION> ;
         <CONTEXT_WITH_OPTION>

   abstract syntax
      chapt  -> CHAPTER_NAME CONTEXT_WITH_OPTION
                INIT ZONE_S ;
      CHAPTER_NAME ::= IDENT string ;
      CONTEXT_WITH_OPTION  ::= CONTEXT void ;
end chapter ;
```

```
chapter TOKENS
   rules
      <ID_INTEGER> ::= <IDENT> ;
         <IDENT>
      <ID_INTEGER> ::= <INT_CONST> ;
         <INT_CONST>
      <PHYLUM> ::= "[ID_INTEGER]" <ID_INTEGER> ;
         <ID_INTEGER>
      <IDENT> ::= %IDENT ;
         ident-atom (%IDENT)
      <IDENT> ::= <META> ;
         <META>
      <PHYLUM> ::= "[IDENT]" <IDENT> ;
         <IDENT>
      <INT_CONST> ::= %INTCONST ;
         int_const-atom (%INTCONST)
      <PHYLUM> ::= "[INT_CONST]" <INT_CONST> ;
         <INT_CONST>
      <TERMINAL> ::= %TERMINAL ;
         terminal-atom (%TERMINAL)
      <STRING> ::= %STRING ;
         string-atom (%STRING)
      <META> ::= %META ;
         meta-atom (%META)
      <METAVARIABLE> ::= <ANONYMOUS_METAVAR> ;
         <ANONYMOUS_METAVAR>
      <METAVARIABLE> ::= <NAMED_METAVAR> ;
         <NAMED_METAVAR>
      <PHYLUM> ::= "[METAVARIABLE]" <METAVARIABLE> ;
         <METAVARIABLE>
      <ANONYMOUS_METAVAR> ::= "*" ;
         anonymous_metavar ()
      <NAMED_METAVAR> ::= <SINGLE_METAVAR> ;
         <SINGLE_METAVAR>
      <NAMED_METAVAR> ::= <SUBLIST_METAVAR> ;
         <SUBLIST_METAVAR>
      <PHYLUM> ::= "[NAMED_METAVAR]" <NAMED_METAVAR> ;
```

```
      <NAMED_METAVAR>
   <SINGLE_METAVAR> ::= "*" <IDENT> ;
      let ident-atom (X) = <IDENT> in
         single_metavar-atom (X)
   <SUBLIST_METAVAR> ::= "**" <IDENT> ;
      let ident-atom (X) = <IDENT> in
         sublist_metavar-atom (X)
   <CONTEXT> ::= <ID_INTEGER> ;
      case <ID_INTEGER>
         when ident-atom (X) | int_const-atom (X)
            => context-atom (X)
      end case
   <PHYLUM> ::= "[CONTEXT]" <CONTEXT> ;
      <CONTEXT>
   <FRAME> ::= "^" <IDENT> ;
      let ident-atom (X) = <IDENT> in
         frame-atom (X)
   <PHYLUM> ::= "[FRAME]" <FRAME> ;
      <FRAME>
   <ARITH_FUNCTION_ARG> ::= <NAMED_METAVAR> ;
      <NAMED_METAVAR>
   <ARITH_FUNCTION_ARG> ::= <FRAME> ;
      <FRAME>
   <PHYLUM> ::= "[ARITH_FUNCTION_ARG]"
                  <ARITH_FUNCTION_ARG> ;
      <ARITH_FUNCTION_ARG>
   <FUNCTION_ARG> ::= <ARITH_FUNCTION_ARG> ;
      <ARITH_FUNCTION_ARG>
   <FUNCTION_ARG> ::= <STRING> ;
      <STRING>
   <PHYLUM> ::= "[FUNCTION_ARG]" <FUNCTION_ARG> ;
      <FUNCTION_ARG>

abstract syntax
   ident  -> ;
   int_const  -> implemented as STRING ;
   terminal  -> implemented as STRING ;
```

```
      string  -> implemented as STRING ;
      anonymous_metavar  -> implemented as SINGLETON ;
      single_metavar  -> ;
      sublist_metavar  -> ;
      context  -> ;
      frame  -> ;
      ID_INTEGER  ::= IDENT INT_CONST ;
      IDENT  ::= ident ;
      INT_CONST  ::= int_const ;
      METAVARIABLE  ::= anonymous_metavar NAMED_METAVAR ;
      NAMED_METAVAR  ::= single_metavar sublist_metavar ;
      CONTEXT  ::= context ;
      FRAME  ::= frame ;
      ARITH_FUNCTION_ARG  ::= NAMED_METAVAR frame ;
      FUNCTION_ARG ::= ARITH_FUNCTION_ARG string ;
end chapter ;

chapter PATTERN
   rules
      <CFHC_PATT> ::= <CONTEXT> "::" <FHC_PATT> ;
         context_patt (<FHC_PATT>, <CONTEXT>)
      <CFHC_PATT> ::= <FHC_PATT> ;
         <FHC_PATT>
      <PHYLUM> ::= "[CFHC_PATT]" <CFHC_PATT> ;
         <CFHC_PATT>
      <FHC_PATT> ::= <HC_PATT> <FRAME_S> ;
         annotated_patt (<HC_PATT>, <FRAME_S>)
      <FHC_PATT> ::= <HC_PATT> ;
         <HC_PATT>
      <PHYLUM> ::= "[FHC_PATT]" <FHC_PATT> ;
         <FHC_PATT>
      <HC_PATT> ::= <C_PATT> "!" <ID_INTEGER> ;
         holoph_patt (<C_PATT>, <ID_INTEGER>)
      <HC_PATT> ::= <C_PATT> ;
         <C_PATT>
      <PHYLUM> ::= "[HC_PATT]" <HC_PATT> ;
         <HC_PATT>
```

```
   <C_PATT> ::= <PATT> "where" <COND_S> ;
      cond_patt (<PATT>, <COND_S>)
   <C_PATT> ::= <PATT> ;
      <PATT>
   <PHYLUM> ::= "[C_PATT]" <C_PATT> ;
      <C_PATT>
   <PATT> ::= <METAVARIABLE> ;
      <METAVARIABLE>
   <PATT> ::= <IDENT> ;
      <IDENT>
   <PATT> ::= <NODE_PATT> ;
      <NODE_PATT>
   <PATT> ::= <ATOM_PATT> ;
      <ATOM_PATT>
   <PHYLUM> ::= "[PATT]" <PATT> ;
      <PATT>
   <FRAME_S> ::= <FRAME> ;
      frame_s-list ((<FRAME>))
   <FRAME_S> ::= <FRAME_S> <FRAME> ;
      frame_s-post (<FRAME_S>, <FRAME>)
   <PHYLUM> ::= "[FRAME_S]" <FRAME_S> ;
      <FRAME_S>
   <COND_S> ::= <BOOL_EXP> ;
      cond_s-list ((<BOOL_EXP>))
   <COND_S> ::= <COND_S> "," <BOOL_EXP> ;
      cond_s-post (<COND_S>, <BOOL_EXP>)
   <PHYLUM> ::= "[COND_S]" <COND_S> ;
      <COND_S>

abstract syntax
   context_patt  -> FHC_PATT CONTEXT ;
   annotated_patt  -> HC_PATT FRAME_S ;
   holoph_patt  -> C_PATT ID_INTEGER ;
   cond_patt  -> PATT COND_S ;
   frame_s  -> FRAME + ... ;
   cond_s  -> BOOL_EXP + ... ;
   CFHC_PATT  ::= FHC_PATT context_patt ;
```

```
   FHC_PATT  ::= HC_PATT annotated_patt ;
   HC_PATT  ::= C_PATT holoph_patt ;
   C_PATT  ::= PATT cond_patt ;
   PATT  ::= METAVARIABLE node_patt ident atom_patt ;
   FRAME_S  ::= frame_s ;
   COND_S  ::= cond_s ;

chapter NODE_PATTERNS
   rules
      <NODE_PATT> ::= <IDENT> "(" <PATT_S> ")" ;
         node_patt (<IDENT>, <PATT_S>)
      <PATT_S> ::= ;
         patt_s-list (())
      <PATT_S> ::= <PATT> ;
         patt_s-list ((<PATT>))
      <PATT_S> ::= <PATT_S> "," <PATT> ;
         patt_s-post (<PATT_S>, <PATT>)
      <PHYLUM> ::= "[PATT_S]" <PATT_S> ;
         <PATT_S>

   abstract syntax
      node_patt  -> IDENT PATT_S ;
      patt_s  -> PATT * ... ;
      PATT_S  ::= patt_s ;
end chapter ;

chapter ATOM_PATTERNS
   rules
      <ATOM_PATT> ::= <IDENT> <ATOM_ARG> ;
         atom_patt (<IDENT>, <ATOM_ARG>)
      <ATOM_ARG> ::= <METAVARIABLE> ;
         <METAVARIABLE>
      <ATOM_ARG> ::= <STRING> ;
         <STRING>
      <PHYLUM> ::= "[ATOM_ARG]" <ATOM_ARG> ;
         <ATOM_ARG>
```

```
      abstract syntax
         atom_patt  -> IDENT ATOM_ARG ;
         ATOM_ARG  ::= METAVARIABLE string ;
   end chapter ;
end chapter ;

chapter 'BOXES, ELEMS and SEPS'
   rules
      <ELEM_S> ::= ;
         elem_s-list (())
      <ELEM_S> ::= <ELEM> <ELEM_S> ;
         elem_s-pre (<ELEM>, <ELEM_S>)
      <PHYLUM> ::= "[ELEM_S]" <ELEM_S> ;
         <ELEM_S>
      <ELEM> ::= <ELEM_WITH_ALLOWED_ATTRIBUTS> ;
         <ELEM_WITH_ALLOWED_ATTRIBUTS>
      <ELEM> ::= <CONTEXT_CALL> ;
         <CONTEXT_CALL>
      <ELEM> ::= <ATTRIBUT_ELEM> ;
         <ATTRIBUT_ELEM>
      <ELEM> ::= <SEP> ;
         <SEP>
      <ELEM> ::= <CASE> ;
         <CASE>
      <ELEM> ::= <IF> ;
         <IF>
      <ELEM> ::= <SUBLIST_ITERATOR> ;
         <SUBLIST_ITERATOR>
      <PHYLUM> ::= "[ELEM]" <ELEM> ;
         <ELEM>

   abstract syntax
      elem_s  -> ELEM * ... ;
      ELEM_S  ::= elem_s ;
      ELEM  ::=
         ELEM_WITH_ALLOWED_ATTRIBUTS CONTEXT_CALL
         attribut_elem
```

```
      sep
      case
      if
      SUBLIST_ITERATOR ;

chapter BOX
   rules
      <BOX> ::= "[" <SEP> <ELEM_S> "]" ;
         box (<SEP>, <ELEM_S>)
      <PHYLUM> ::= "[BOX]" <BOX> ;
         <BOX>

   abstract syntax
      box  -> SEP ELEM_S ;
      BOX  ::= box ;
end chapter ;

chapter ATTRIBUTS
   rules
      <ATTRIBUT_ELEM> ::=
         <ATTRIBUTS> <ELEM_WITH_ALLOWED_ATTRIBUTS> ;
         attribut_elem
            (<ATTRIBUTS>, <ELEM_WITH_ALLOWED_ATTRIBUTS>)
      <ATTRIBUTS> ::= "in" <ATTRIBUT_S> ":" ;
         <ATTRIBUT_S>
      <ATTRIBUTS> ::= "in" "term" <ATTRIBUT_S> ":" ;
         let attribut_s-list (X) = <ATTRIBUT_S> in
            local_attribut_s-list (X)
      <ATTRIBUT_S> ::= <ATTRIBUT> ;
         attribut_s-list ((<ATTRIBUT>))
      <ATTRIBUT_S> ::= <ATTRIBUT_S> "," <ATTRIBUT> ;
         attribut_s-post (<ATTRIBUT_S>, <ATTRIBUT>)
      <PHYLUM> ::= "[ATTRIBUT_S]" <ATTRIBUT_S> ;
         <ATTRIBUT_S>
      <ATTRIBUT> ::= <IDENT> "=" <ID_INTEGER> ;
         attribut (<IDENT>, <ID_INTEGER>)
      <PHYLUM> ::= "[ATTRIBUT]" <ATTRIBUT> ;
```

```
         <ATTRIBUT>
      <ELEM_WITH_ALLOWED_ATTRIBUTS> ::= <BOX> ;
         <BOX>
      <ELEM_WITH_ALLOWED_ATTRIBUTS> ::= <TERMINAL> ;
         <TERMINAL>
      <ELEM_WITH_ALLOWED_ATTRIBUTS> ::=
                                 <PP_FUNCTION_CALL> ;
         <PP_FUNCTION_CALL>
      <PHYLUM> ::=
         "[ELEM_WITH_ALLOWED_ATTRIBUTS]"
            <ELEM_WITH_ALLOWED_ATTRIBUTS> ;
         <ELEM_WITH_ALLOWED_ATTRIBUTS>

   abstract syntax
      attribut_elem  ->
         ATTRIBUT_S ELEM_WITH_ALLOWED_ATTRIBUTS ;
      attribut_s  -> ATTRIBUT + ... ;
      attribut  -> IDENT ID_INTEGER ;
      local_attribut_s  -> ATTRIBUT + ... ;
      ATTRIBUT_S  ::= attribut_s local_attribut_s ;
      ATTRIBUT  ::= attribut ;
      ELEM_WITH_ALLOWED_ATTRIBUTS  ::=
         box terminal pp_fn_call ;
end chapter ;

chapter RECURSIVE_CALL
   rules
      <CONTEXT_CALL> ::= <CONTEXT> "::" <HOLOPH_CALL> ;
         context_call (<HOLOPH_CALL>, <CONTEXT>)
      <CONTEXT_CALL> ::= <HOLOPH_CALL> ;
         <HOLOPH_CALL>
      <PHYLUM> ::= "[CONTEXT_CALL]" <CONTEXT_CALL> ;
         <CONTEXT_CALL>
      <HOLOPH_CALL> ::= <SIMPLE_CALL> "!" <HOLOPH_EXP> ;
         holoph_call (<SIMPLE_CALL>, <HOLOPH_EXP>)
      <HOLOPH_CALL> ::= <SIMPLE_CALL> ;
         <SIMPLE_CALL>
```

```
      <PHYLUM> ::= "[HOLOPH_CALL]" <HOLOPH_CALL> ;
         <HOLOPH_CALL>
      <SIMPLE_CALL> ::= <NAMED_METAVAR> ;
         <NAMED_METAVAR>
      <SIMPLE_CALL> ::= <FRAME> ;
         <FRAME>
      <PHYLUM> ::= "[SIMPLE_CALL]" <SIMPLE_CALL> ;
         <SIMPLE_CALL>
      <HOLOPH_EXP> ::= <V_IDENT> "-" <INT_CONST> ;
         minus (<V_IDENT>, <INT_CONST>)
      <HOLOPH_EXP> ::= <V_IDENT> "+" <INT_CONST> ;
         plus (<V_IDENT>, <INT_CONST>)
      <HOLOPH_EXP> ::= <ID_INTEGER> ;
         <ID_INTEGER>
      <PHYLUM> ::= "[HOLOPH_EXP]" <HOLOPH_EXP> ;
         <HOLOPH_EXP>
      <V_IDENT> ::= ;
         void ()
      <V_IDENT> ::= <IDENT> ;
         <IDENT>
      <PHYLUM> ::= "[V_IDENT]" <V_IDENT> ;
         <V_IDENT>

   abstract syntax
      context_call  -> HOLOPH_CALL CONTEXT ;
      holoph_call  -> SIMPLE_CALL HOLOPH_EXP ;
      plus  -> V_IDENT INT_CONST ;
      minus  -> V_IDENT INT_CONST ;
      CONTEXT_CALL  ::= HOLOPH_CALL context_call ;
      HOLOPH_CALL  ::= SIMPLE_CALL holoph_call ;
      SIMPLE_CALL  ::= NAMED_METAVAR frame ;
      HOLOPH_EXP  ::= ID_INTEGER plus minus ;
      V_IDENT  ::= IDENT void ;
end chapter ;

chapter SEPARATORS
   rules
```

```
            <SEP> ::= "<" <IDENT> <S_PARAM_S> ">" ;
               sep (<IDENT>, <S_PARAM_S>)
            <PHYLUM> ::= "[SEP]" <SEP> ;
               <SEP>
            <S_PARAM_S> ::= ;
               s_param_s-list (())
            <S_PARAM_S> ::= <ID_INTEGER> ;
               s_param_s-list ((<ID_INTEGER>))
            <S_PARAM_S> ::= <S_PARAM_S> "," <ID_INTEGER> ;
               s_param_s-post (<S_PARAM_S>, <ID_INTEGER>)
            <PHYLUM> ::= "[S_PARAM_S]" <S_PARAM_S> ;
               <S_PARAM_S>

         abstract syntax
            sep  -> IDENT S_PARAM_S ;
            s_param_s  -> ID_INTEGER * ... ;
            SEP  ::= sep ;
            S_PARAM_S  ::= s_param_s ;
      end chapter ;
   end chapter ;

   chapter CONDITIONNAL_COMMAND_S
      chapter CASE
         rules
            <CASE> ::=
               "case" <SELECTOR> "of" <ALTERNATIVE_S> "end"
                  "case" ;
               case (<SELECTOR>, <ALTERNATIVE_S>)
            <SELECTOR> ::= <NAMED_METAVAR> ;
               <NAMED_METAVAR>
            <SELECTOR> ::= <ARITH_FUNCTION_CALL> ;
               <ARITH_FUNCTION_CALL>
            <SELECTOR> ::= <IDENT> ;
               <IDENT>
            <PHYLUM> ::= "[SELECTOR]" <SELECTOR> ;
               <SELECTOR>
            <ALTERNATIVE_S> ::= <ALTERNATIVE> ;
```

```
      alternative_s-list ((<ALTERNATIVE>))
   <ALTERNATIVE_S> ::= <ALTERNATIVE_S> <ALTERNATIVE> ;
      alternative_s-post (<ALTERNATIVE_S>,
                          <ALTERNATIVE>)
   <PHYLUM> ::= "[ALTERNATIVE_S]" <ALTERNATIVE_S> ;
      <ALTERNATIVE_S>
   <ALTERNATIVE> ::= <CHOICE_S> ":" <ELEM_S> ";" ;
      alternative (<CHOICE_S>, <ELEM_S>)
   <PHYLUM> ::= "[ALTERNATIVE]" <ALTERNATIVE> ;
      <ALTERNATIVE>
   <CHOICE_S> ::= <CHOICE> ;
      choice_s-list ((<CHOICE>))
   <CHOICE_S> ::= <CHOICE_S> "," <CHOICE> ;
      choice_s-post (<CHOICE_S>, <CHOICE>)
   <PHYLUM> ::= "[CHOICE_S]" <CHOICE_S> ;
      <CHOICE_S>
   <CHOICE> ::= <PATT> ;
      <PATT>
   <CHOICE> ::= <INT_CONST> ;
      <INT_CONST>
   <CHOICE> ::= "others" ;
      others ()
   <PHYLUM> ::= "[CHOICE]" <CHOICE> ;
      <CHOICE>

abstract syntax
   case  -> SELECTOR ALTERNATIVE_S ;
   alternative_s  -> ALTERNATIVE + ... ;
   choice_s  -> CHOICE + ... ;
   alternative  -> CHOICE_S ELEM_S ;
   others  -> implemented as SINGLETON ;
   SELECTOR  ::=
      NAMED_METAVAR arith_function_call ident ;
   ALTERNATIVE_S  ::= alternative_s ;
   ALTERNATIVE  ::= alternative ;
   CHOICE_S  ::= choice_s ;
   CHOICE  ::= PATT INT_CONST others ;
```

```
end chapter ;

chapter IF_COMMAND
   rules
      <IF> ::=
         "if" <BOOL_EXP> "then" <ELEM_S> <ELSIF_OPTION>
            "end"
            "if" ;
         if-pre (cond_clause (<BOOL_EXP>, <ELEM_S>),
               <ELSIF_OPTION>)
      <IF> ::=
         "if" <BOOL_EXP> "then" <ELEM_S> <ELSIF_OPTION>
            "else"
            <ELEM_S>
            "end"
            "if" ;
         if-pre (cond_clause (<BOOL_EXP>, <ELEM_S>.1),
               if-post (<ELSIF_OPTION>,
                     cond_clause (void (), <ELEM_S>.2)))
      <ELSIF_OPTION> ::=
         <ELSIF_OPTION> "elsif" <BOOL_EXP>
                        "then" <ELEM_S> ;
         if-post (<ELSIF_OPTION>,
               cond_clause (<BOOL_EXP>, <ELEM_S>))
      <ELSIF_OPTION> ::= ;
         if-list (())
      --********* No entry points given here.


   abstract syntax
      if  -> COND_CLAUSE + ... ;
      cond_clause  -> BOOL_EXP ELEM_S ;
      COND_CLAUSE  ::= cond_clause ;

   chapter BOOL_EXP
      rules
         <BOOL_EXP> ::= <BOOL_EXP_1> ;
```

```
   <BOOL_EXP_1>
<BOOL_EXP> ::= <BOOL_EXP_1> "and" <BOOL_EXP_1> ;
   and (<BOOL_EXP_1>.1, <BOOL_EXP_1>.2)
<BOOL_EXP> ::= <BOOL_EXP_1> "or" <BOOL_EXP_1> ;
   or (<BOOL_EXP_1>.1, <BOOL_EXP_1>.2)
<PHYLUM> ::= "[BOOL_EXP]" <BOOL_EXP> ;
   <BOOL_EXP>
<BOOL_EXP_1> ::= "not" <BOOL_EXP_1> ;
   not (<BOOL_EXP_1>)
<BOOL_EXP_1> ::= "(" <BOOL_EXP> ")" ;
   <BOOL_EXP>
<BOOL_EXP_1> ::= <S_BOOL_EXP> ;
   <S_BOOL_EXP>
<S_BOOL_EXP> ::=
   <ARITH_EXP> <ARITH_OP> <ARITH_EXP> ;
   arith_bool
      (<ARITH_EXP>.1, <ARITH_OP>, <ARITH_EXP>.2)
<S_BOOL_EXP> ::= <METAVAROP_S> "in" <SORT_S> ;
   patt_cond (<METAVAROP_S>, <SORT_S>)
<METAVAROP_S> ::= <NAMED_METAVAR> ;
   metavarop_s-list ((<NAMED_METAVAR>))
<METAVAROP_S> ::= <METAVAROP_S> ","
                     <NAMED_METAVAR> ;
   metavarop_s-post
        (<METAVAROP_S>, <NAMED_METAVAR>)
<PHYLUM> ::= "[METAVAROP_S]" <METAVAROP_S> ;
   <METAVAROP_S>
<ARITH_OP> ::= "=" ;
   equal ()
<ARITH_OP> ::= ">" ;
   greater ()
<ARITH_OP> ::= ">=" ;
   geq ()
<ARITH_OP> ::= "<" ;
   lesser ()
<ARITH_OP> ::= "<=" ;
   leq ()
```

```
            <PHYLUM> ::= "[ARITH_OP]" <ARITH_OP> ;
               <ARITH_OP>
      end chapter ;

      abstract syntax
         arith_bool  -> ARITH_EXP ARITH_OP ARITH_EXP ;
         equal  -> implemented as SINGLETON ;
         greater  -> implemented as SINGLETON ;
         geq  -> implemented as SINGLETON ;
         lesser  -> implemented as SINGLETON ;
         leq  -> implemented as SINGLETON ;
         not  -> BOOL_EXP ;
         and  -> BOOL_EXP BOOL_EXP ;
         or  -> BOOL_EXP BOOL_EXP ;
         patt_cond  -> METAVAROP_S SORT_S ;
         metavarop_s  -> NAMED_METAVAR + ... ;
         BOOL_EXP  ::= void arith_bool patt_cond
                       not and or ;
         METAVAROP_S  ::= metavarop_s ;
         ARITH_OP  ::= equal greater geq lesser leq ;
   end chapter ;
end chapter ;

chapter ATOM_PP
   rules
      <PP_FUNCTION_CALL> ::= <IDENT>
                             "(" <FUNCTION_ARG> ")" ;
         pp_fn_call (<IDENT>, <FUNCTION_ARG>)

   abstract syntax
      pp_fn_call  -> IDENT FUNCTION_ARG ;
end chapter ;

chapter LIST_PP
   rules
      <SUBLIST_ITERATOR> ::= "(" <ELEM_S> ")" ;
         seplist (<ELEM_S>)
```

```
         <SUBLIST_ITERATOR> ::= "\\" "(" <ELEM_S> ")" ;
            revlist (<ELEM_S>)
         <PHYLUM> ::= "[SUBLIST_ITERATOR]" <SUBLIST_ITERATOR> ;
            <SUBLIST_ITERATOR>

      abstract syntax
         seplist  -> ELEM_S ;
         revlist  -> ELEM_S ;
         SUBLIST_ITERATOR  ::= seplist revlist ;
   end chapter ;

   abstract syntax
      void  -> implemented as SINGLETON ;
end definition
```

7 Ppml primitives

The Ppml compiler is not loaded automatically when you run Centaur. It is loaded automatically when you try to compile a Ppml program. You may also load it by hand with the call:

```
(loadmodule 'centaur/sources/comp_ppml)
```

Subsequently, its presence may be detected with the call
`(module-loaded-p 'centaur/sources/comp_ppml)`.

Please consult the Centaur resource manual for details about the location of generated and read pretty printer modules.

7.1 Compiling a Ppml program

({ppml}:compile program^{tree} **module-data**$^{a\text{-}list}$ [**mode**]^{symbol} ***)**
→ *boolean*

This function compiles a complete Ppml specification and returns **t** if compilation succeeds, otherwise **()**. In order to compile a program, the language's tables must be present in memory.

When present the **mode** arguments determine the effect of the compilation. Possible values are:

- **interp** : For a formalism `L` and pretty printer `pp`, compiles the **tree** and generates a file named `L-pp.ll`.
- **module** : Compiles the **tree** and generates the `L-pp.ll` file and a file named `module.LM` that serves to create a pretty printer module.
- **large** : Due to a Le-Lisp "feature," large modules must export all generated functions. If you specify this mode, you must also specify the **module** mode.
- **non-compact** : Pretty prints the contents of generated `.ll` file in a readable form.

The **module-data** argument provides information that guides the creation of the `module.LM` file. It is an a-list whose keys may be the following, with the meaning of their values:

- **name** : The name of module to appear in the `module.LM` file.
- **files** : A list of files to be included in the module. This list must contain *all* files, including `L-pp.ll` and `L-pp.at`. File names appear as is in `module.LM`. File path names are relative to a root directory.
- **import** : Modules to be imported for this pretty printer module. The `compile` function automatically includes those modules required by the pretty printer itself.
- **export** : Functions to be exported by this module. The `compile` function automatically exports those functions generated by the compilation.

This function also cleans up the lisp environment (using the function {centaur}:remove-pprinter) so that the pretty printer is loaded the next time a tree in the formalism is to be pretty printed with it.

7.2 Loading a pretty printer

({centaur}:load-pprinter formalism-name^{symbol} **pp-name**^{symbol} **[load_flag]**^{boolean}**)** $\rightarrow$ *pprinter-list*

This function loads the pretty printer **pp-name** into the Lisp environment for the formalism **formalism-name** and returns the list of loaded pretty printers. If load_flag is present, it removes all information pertaining to this pretty printer before reloading, otherwise it loads the pretty printer (unless it has already been loaded). This function also Loads the `.ll` and `.at` files for the indicated formalism and pretty printer. **N.B.** Since this function modifies several variables in the Lisp environment, you should use it to load a `.ll` file instead of Lisp's load functions. A printed message indicates whether the pretty printer was loaded or if it wasn't found.

7.3 Removing a pretty printer

({centaur}:remove-pprinter formalism-name^{symbol}
pprinter-name^{symbol}**)** $\rightarrow$ *pprinter-list*

This function removes a pretty printer from the CENTAUR environment. It returns the modified list of known pretty printers.

8 Function index

9 Error index