

# CafeOBJ Syntax Quick Reference

for Interpreter version 1.4.8 or later

Toshimi Sawada\*  
Kosakusha Inc.

September 27, 2019

## Contents

<b>1</b>	<b>Syntax</b>	<b>1</b>
1.1	CafeOBJ Spec . . . . .	2
1.2	Module Declaration . . . . .	2
1.3	Module Expression . . . . .	3
1.4	View Declaration . . . . .	3
1.5	Evaluation . . . . .	3
1.6	Sugars and Abbreviations . . . . .	3
<b>2</b>	<b>Lexical Considerations</b>	<b>5</b>
2.1	Reserved Word . . . . .	6
2.2	Self-terminating Characters . . . . .	6
2.3	Identifier . . . . .	6
2.4	Operator Symbol . . . . .	6
2.5	Comments and Separators . . . . .	6

## 1 Syntax

We use an extended BNF grammar to define the syntax. The general form of a production is

$$\textit{nonterminal} ::= \textit{alternative} \mid \textit{alternative} \mid \cdots \mid \textit{alternative}$$

The following extensions are used:

$a \cdots$	a list of one or more $as$ .
$a, \cdots$	a list of one or more $as$ separated by commas: “ $a$ ” or “ $a, a$ ” or “ $a, a, a$ ”, etc.
$\{ a \}$	$\{$ and $\}$ are meta-syntactical brackets treating $a$ as one syntactic category.
$[ a ]$	an optional $a$ : “ $ $ ” or “ $a$ ”.

Nonterminal symbols appear in *italic face*. Terminal symbols appear in the face like this: “terminal”, and may be surrounded by “ and ” for emphasis or to avoid confusion with meta characters used in the extended BNF. We will refer terminal symbols other than self-terminating characters (see section 2.2) as *keywords* in this document.

---

\*sawada@cafeobj.org

## 1.1 CafeOBJ Spec

$spec ::= \{ module \mid view \mid eval \} \dots$

A CafeOBJ spec is a sequence of *module* (module declaration – section 1.2), *view* (view declaration – section 1.4) or *eval* (*reduce* or *execute* term – section 1.5).

## 1.2 Module Declaration

<i>module</i>	$::= module\_type \ module\_name \ [ \ parameters \ ] \ [ \ principal\_sort \ ]$ $\quad \{ " \ module\_elt \ \dots \ " \}$	
<i>module_type</i>	$::= module \mid module! \mid module^*$	– 1
<i>module_name</i>	$::= ident$	
<i>parameters</i>	$::= "( \ parameter, \dots \ )"$	
<i>parameter</i>	$::= [ \ protecting \mid extending \mid including \ ] \ paramter\_name \ :: \ module\_expr$	– 23
<i>parameter_name</i>	$::= ident$	
<i>principal_sort</i>	$::= principal\_sort \ sort\_name$	
<i>module_elt</i>	$::= import \mid sort \mid operator \mid variable \mid axiom \mid macro \mid comment$	– 4
<i>import</i>	$::= \{ \ protecting \mid extending \mid including \mid using \} "( \ module\_expr \ )"$	
<i>sort</i>	$::= visible\_sort \mid hidden\_sort$	
<i>visible_sort</i>	$::= "[ \ sort\_decl, \dots \ ]"$	
<i>hidden_sort</i>	$::= "[ * [ \ sort\_decl, \dots \ ] ]"$	
<i>sort_decl</i>	$::= sort\_name \ \dots \ [ \ supersorts \ \dots \ ]$	
<i>supersorts</i>	$::= < \ sort\_name \ \dots$	
<i>sort_name</i>	$::= sort\_symbol [ \ qualifier \ ]$	– 5
<i>sort_symbol</i>	$::= ident$	
<i>qualifier</i>	$::= ". \ module\_name$	
<i>operator</i>	$::= \{ \ op \mid bop \} \ operator\_symbol \ : \ [ \ arity \ ] \ \rightarrow \ coarity \ [ \ op\_attrs \ ]$	– 6
<i>arity</i>	$::= sort\_name \ \dots$	
<i>coarity</i>	$::= sort\_name$	
<i>op_attrs</i>	$::= "( \ op\_attr \ \dots \ )"$	
<i>op_attr</i>	$::= constr \mid associative \mid commutative \mid idempotent \mid \{ \ id: \mid idr: \} "( \ term \ )"$ $\mid \text{strat: } "( \ natural \ \dots \ )" \mid \text{prec: } natural \mid \text{l-assoc} \mid \text{r-assoc} \mid \text{coherent} \mid \text{demod}$	– 7
<i>variable</i>	$::= var \ var\_name \ : \ sort\_name \mid vars \ var\_name \ \dots \ : \ sort\_name$	
<i>var_name</i>	$::= ident$	
<i>axiom</i>	$::= equation \mid cequation \mid transition \mid ctransition \mid fol$	
<i>equation</i>	$::= \{ \ eq \mid beq \} [ \ label \ ] \ term = term \ ."$	
<i>cequation</i>	$::= \{ \ ceq \mid bceq \} [ \ label \ ] \ term = term \text{ if } term \ ."$	
<i>transition</i>	$::= \{ \ trans \mid btrans \} [ \ label \ ] \ term \Rightarrow term \ ."$	
<i>ctransition</i>	$::= \{ \ ctrans \mid bctrans \} [ \ label \ ] \ term \Rightarrow term \text{ if } term \ ."$	
<i>fol</i>	$::= ax [ \ label \ ] \ term \ ."$	
<i>label</i>	$::= "[ \ ident \ \dots \ ]:"$	
<i>macro</i>	$::= \#define \ term \ ::= term \ ."$	

<sup>1</sup>The nonterminal *ident* is for identifiers and will be defined in the section 2.3.

<sup>2</sup>*module\_expr* is defined in the section 1.3.

<sup>3</sup>If optional  $[ \ protecting \mid extending \mid including \ ]$  is omitted, it is defaulted to protecting.

<sup>4</sup>*comment* is discussed in section 2.5.

<sup>5</sup>There must not be any separators (see section 2) between *ident* and *qualifier*.

<sup>6</sup>*operator\_symbol* is defined in section 2.4.

<sup>7</sup>*natural* is a natural number written in ordinal arabic notation.

### 1.3 Module Expression

```

module_expr ::= module_name | sum | rename | instantiation | ("module_expr")
sum          ::= module_expr { + module_expr } ...
rename       ::= module_expr * {"rename_map, ..."}
instantiation ::= module_expr ("{" ident[qualifier] <= aview}, ... ")
rename_map   ::= sort_map | op_map
sort_map     ::= { sort | hsort } sort_name -> ident
op_map       ::= { op | bop } op_name -> operator_symbol
op_name      ::= operator_symbol | ("operator_symbol")qualifier
aview        ::= view_name | module_expr
              | view to module_expr {"view_elt, ..."}
view_name    ::= ident
view_elt     ::= sort_map | op_view | variable
op_view      ::= op_map | term -> term

```

When a module expression is not fully parenthesized, the proper nesting of subexpressions may be ambiguous. The following precedence rule is used to resolve such ambiguity:

$$sum < rename < instantiation$$

### 1.4 View Declaration

```

view ::= view view_name from module_expr to module_expr
      {" view_elt, ... "}

```

### 1.5 Evaluation

```

eval    ::= { reduce | behavioural-reduce | execute } context term "."
context ::= in module_expr :

```

The interpreter has a notion of *current module* which is specified by a *module\_expr* and establishes a context. If it is set, *context* can be omitted.

### 1.6 Sugars and Abbreviations

**Module type** There are following abbreviations for *module\_type*.

Keyword	Abbreviation
module	mod
module!	mod!
module*	mod*

#### Module Declaration

```
make ::= make module_name (" module_expr ")
```

*make* is a short hand for declaring module of name *module\_name* which imports *module\_expr* with protecting mode.  
 make FOO (BAR \* {sort Bar -> Foo})

is equivalent to

```
module FOO { protecting (BAR * {sort Bar -> Foo}) }
```

**Principal Sort** principal-sort can be abbreviated to psort.

**Import Mode** For import modes, the following abbreviations can be used:

Keyword	Abbreviation
protecting	pr
extending	ex
including	inc
using	us

**Simultaneous Operator Declaration** Several operators with the same arity, coarity and operator attributes can be declared at once by ops. The form

$\text{ops } \text{operator\_symbol}_1 \cdots \text{operator\_symbol}_n : \text{arity} \rightarrow \text{coarity } \text{op\_attrs}$

is just equivalent to the following multiple operator declarations:

$\text{op } \text{operator\_symbol}_1 : \text{arity} \rightarrow \text{coarity } \text{op\_attrs}$

$\vdots$

$\text{op } \text{operator\_symbol}_n : \text{arity} \rightarrow \text{coarity } \text{op\_attrs}$

bops is the counterpart of ops for behavioural operators.

$\text{bops } \text{operator\_symbol} \cdots : \text{arity} \rightarrow \text{coarity } \text{op\_attrs}$

In simultaneous declarations, parentheses are sometimes necessary to separate operator symbols. This is always required if an operator symbol contains dots, blank characters or underscores.

**Predicate** Predicate declaration (*predicate*) is a syntactic sugar for declaring Bool valued operators, and has the syntax:

$\text{predicate} ::= \text{pred } \text{operator\_symbol} : \text{arity} [ \text{op\_attrs} ] \quad -^8$

The form

$\text{pred } \text{operator\_symbol} : \text{arity } \text{op\_attrs}$

is equivalent to:

$\text{op } \text{operator\_symbol} : \text{arity} \rightarrow \text{Bool } \text{op\_attrs}$

**Operator Attributes** The following abbreviations are available:

Keyword	Abbreviation
associative	assoc
commutative	comm
idempotent	idem

---

<sup>8</sup>You cannot use *sort\_name* of the same character sequence as that of any keywords, i.e., module, op, vars, etc. in *arity*.

**Axioms** For the keywords introducing axioms, the following abbreviations can be used:

Keyword	Abbreviation	Keyword	Abbreviation
ceq	cq	bceq	bcq
trans	trns	ctrans	ctrns
btrans	btrns	bctrans	bctrns

**Blocks of Declarations** References to (importations of) other modules, signature definitions and axioms can be clustered in blocked declarations:

```

imports ::= imports "{"
           { import | comment } ...
           "}"
signature ::= signature "{"
           { sort | record | operator | comment } ...
           "}"
axioms ::= axioms "{"
           { variable | axiom | comment } ...
           "}"

```

**Views** To reduce the complexity of views appearing in module instantiation, some sugars are provided.

First, it is possible to identify parameters by positions, not by names. For example, if a parameterized module is declared like

```
module! FOO (A1 :: TH1, A2 :: TH2) { ... }
```

the form

```
FOO(V1, V2)
```

is equivalent to

```
FOO(A1 <= V1, A2 <= V2)
```

Secondly, view to construct in arguments of module instantiations can always be omitted. That is,

```
FOO(A1 <= view to module_expr{...})
```

can be written as

```
FOO(A1 <= module_expr{...})
```

## Evaluation

Keyword	Abbreviation
reduce	red
bereduce	bred
execute	exec

## 2 Lexical Considerations

A CafeOBJ spec is written as a sequence of tokens and separators. A *token* is a sequence of “printing” ASCII characters (octal 40 through 176).<sup>9</sup> A *separator* is a “blank” character (space, vertical tab, horizontal tab, carriage return, newline, form feed). In general, any number of separators may appear between tokens.

<sup>9</sup>The current interpreter accepts Unicode characters also, but this is beyond the definition of CafeOBJ language.

## 2.1 Reserved Word

There are *no* reserved word in CafeOBJ. One can use keywords such as `module`, `op`, `var`, or `signature`, etc. for identifiers or operator symbols.

## 2.2 Self-terminating Characters

The following eight characters are always treated as *self-terminating*, i.e., the character itself construct a token.

( ) , [ ] { } ;

## 2.3 Identifier

Nonterminal *ident* is for *identifier* which is a sequence of any printing ASCII characters except the followings:

self-terminating characters (see section 2.2)  
. (dot)  
" (double quote)

Upper- and lowercase are distinguished in identifiers. *idents* are used for module names (*module\_name*), view names (*view\_name*), parameter names (*parameter\_name*), sort symbols (*sort\_symbol*), variables(*var\_name*), slot names (*slot\_name*) and labels (*label*).

## 2.4 Operator Symbol

The nonterminal *operator\_symbol* is used for naming operators (*operator*) and is a sequence of any ASCII characters (self-terminating characters or non-printing characters can be an element of operator names.)<sup>10</sup>

Underscores are specially treated when they apper as a part of operator names; they reserve the places where arguments of the operator are inserted. Thus the single underscore cannot be a name of an operator.

## 2.5 Comments and Separators

A *comment* is a sequence of characters that begins with one of the following four character sequences

--    -->  
\*\*    \*\*>

which ends with a newline character, and contains only printing ASCII characters and horizontal tabs in between.

A *separator* is a blank character (space, vertical tab, horizontal tab, carriage return, newline, from feed). One or more separators must appear between any two adjuacent non-self-terminating tokens.<sup>11</sup>

Comments also act as separators, but their apperance is limited to some specific places (see section 1).

---

<sup>10</sup>The current implementation does not allow EOT character (control-D) to be an element of operator symbol.

<sup>11</sup>The same rule is applied to *term*. Further, if an *operator\_symbol* contains blanks or self-terminating characters, it is sometimes neccessary to enclose a term with such operator as top by parentheses for disambiguation.