CafeOBJ Syntax Quick Reference

for Interpreter version 1.4.8

Toshimi Sawada
Software Research Associates, Inc.

June 9, 2010

Contents

1 Syntax **1**
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 Abbriviations . . . 3

2 Lexical Considerations **5**
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

$$nonterminal ::= alternative \mid alternative \mid \cdots \mid alternative$$

The following extensions are used:

a $\cdots$	a list of one or more *a*s.
a, $\cdots$	a list of one or more *a*s 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 *keyword*s in this document.

1.1 CafeOBJ Spec

$spec$ ::= { $module$ | $view$ | $eval$ } $\cdots$

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

$module$	::= $module\_type$ $module\_name$ [$parameters$] [$principal\_sort$] "{" $module\_elt \cdots$ "}"	
$module\_type$	::= module | module! | module*	
$module\_name$	::= $ident$	– [1]
$parameters$	::= "(" $parameter, \cdots$ ")"	
$parameter$	::= [protecting | extending | including] $paramter\_name$ :: $module\_expr$	– [2][3]
$parameter\_name$	::= $ident$	
$principal\_sort$	::= principal-sort $sort\_name$	
$module\_elt$	::= $import$ | $sort$ | $operator$ | $variable$ | $axiom$ | $comment$	– [4]
$import$	::= { protecting | extending | including | using }"(" $module\_expr$ ")"	
$sort$	::= $visible\_sort$ | $hidden\_sort$	
$visible\_sort$	::= "[" $sort\_decl, \cdots$ "]"	
$hidden\_sort$	::= "*[" $sort\_decl, \cdots$ "]*"	
$sort\_decl$	::= $sort\_name$ $\cdots$ [$supersorts$ $\cdots$]	
$supersorts$	::= < $sort\_name$ $\cdots$	
$sort\_name$	::= $sort\_symbol$[$qualifier$]	– [5]
$sort\_symbol$	::= $ident$	
$qualifier$	::= "."$module\_expr$[$qualifier$]	
$operator$	::= { op | bop } $operator\_symbol$: [$arity$] -> $coarity$ [op_attrs]	– [6]
$arity$	::= $sort\_name \cdots$	
$coarity$	::= $sort\_name$	
op_attrs	::= "{" $op\_attr \cdots$ "}"	
op_attr	::= constr | associative | commutative | idempotent | { id: | idr: }"(" $term$ ")" | strat: "(" $natural \cdots$ "")" | prec: $natural$ | l-assoc | r-assoc | coherent	– [7]
$variable$	::= var var_name : $sort\_name$ | vars $var\_name \cdots$: $sort\_name$	
var_name	::= $ident$	
$axiom$	::= $equation$ | $cequation$ | $transition$ | $ctransition$	
$equation$	::= { eq | beq } [$label$] $term$ = $term$ "."	
$cequation$	::= { ceq | bceq } [$label$] $term$ = $term$ if $term$ "."	
$transition$	::= { trans | btrans } [$label$] $term$ => $term$ "."	
$ctransition$	::= { ctrans | bctrans } [$label$] $term$ => $term$ if $term$ "."	
$label$	::= "[" $ident$ "]:"	

[1] The nonterminal *ident* is for identifiers and will be defined in the section 2.3.
[2] *module_expr* is defined in the section 1.3.
[3] If optional [protecting | extending | including] is omitted, it is defaulted to protecting.
[4] *comment* is descussed in section 2.5.
[5] There must not be any separators (see section 2) between *ident* and *qualifier*.
[6] *operator_symbol* is defined in section 2.4.
[7] *natural* is a natural number written in ordinal arabic notation.

1.3 Module Expression

$module\_expr$ ::= $module\_name$ | sum | $rename$ | $instantiation$ | "("$module\_expr$")"
sum ::= $module\_expr$ { + $module\_expr$ } $\cdots$
$rename$ ::= $module\_expr$ * "{"$rename\_map, \cdots$"}"
$instantiation$ ::= $module\_expr$ "("{ $ident[qualifier]$ <= $aview$}, $\cdots$ ")"
$rename\_map$ ::= $sort\_map$ | op_map
$sort\_map$ ::= { sort | hsort } $sort\_name$ -> $ident$
op_map ::= { op | bop } op_name -> $operaotr\_symbol$
op_name ::= $operator\_symbol$ | "("$operator\_symbol$")"$qualifier$
$aview$ ::= $view\_name$ | $module\_expr$
| view to $module\_expr$ "{"$view\_elt, \cdots$"}"
$view\_name$ ::= $ident$
$view\_elt$ ::= $sort\_map$ | 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 $module\_expr$
"{" $view\_elt, \cdots$ "}"

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 Abbriviations

Module type There are following abbreviations for *module_type*.

Keyword	Abbriviation
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 abbriviated to psort.

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

Keyword	Abbriviation
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

ops $operator\_symbol_1 \cdots operator\_symbol_n$: $arity$ -> $coarity$ op_attrs

is just equivalent to the following multiple operator declarations:

op $operator\_symbol_1$: $arity$ -> $coarity$ op_attrs
$\vdots$
op $operator\_symbol_n$: $arity$ -> $coarity$ op_attrs

bops is the counterpart of ops for behavioural operators.

bops $operator\_symbol$ $\cdots$: $arity$ -> $coarity$ 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:

$predicate$ ::= pred $operator\_symbol$: $arity$ [op_attrs] – [8]

The form

pred $operator\_symbol$: $arity$ op_attrs

is equivalent to:

op $operator\_symbol$: $arity$ -> Bool op_attrs

Operator Attributes The following abbriviations are available:

Keyword	Abbriviation
associative	assoc
commutative	comm
idempotent	idem

[8] 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 abbriviations can be used:

Keyword	Abbriviation	Keyword	Abbriviation
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 clusterd in blocked declarations:

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

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	Abbriviation
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).[9] A *separator* is a "blank" character (space, vertical tab, horizontal tab, carriage return, newline, form feed). In general, any mumber of separators may appear between tokens.

[9]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 seven 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 sequnce 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. *ident*s 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.)[10]

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.[11]

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

[10] The current implementation does not allow EOT character (control-D) to be an element of operator symbol.

[11] 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.