CafeOBJ Syntax Quick Reference

for Interpreter version 1.5.0 or later

Toshimi Sawada

February 29, 2016

Contents

1 Syntax
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

2 Lexical Considerations
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

\[ \text{nonterminal} ::= \text{alternative} | \text{alternative} | \cdots | \text{alternative} \]

The following extensions are used:

- \(a^*\) a list of one or more \(a\).
- \(a, \cdots\) a list of one or more \(a\) 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.
1.1 CafeOBJ Spec

spec ::= { module | view | eval } ···

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 1.3).

1.2 Module Declaration

module ::= module_type module_name [ parameters ] [ principal_sort ]
    {"module_elts ... "}
module_type ::= module | module! | module*
module_name ::= ident
parameters ::= {"parameter, ... "}
parameter ::= { protecting | extending | including } parameter_name : module_expr
parameter_name ::= ident
principal_sort ::= principal-sort sort_name
module_elts ::= import | sort | operator | variable | axiom | macro | comment
import ::= { protecting | extending | including | using } "(" module_expr ")"
sort ::= visible_sort | hidden_sort
visible_sort ::= ["sort_decl, ... "]
hidden_sort ::= ["(" sort_decl, ... ")"]
sort_decl ::= sort_name ... [ supersorts ... ]
supersorts ::= < sort_name ...>
sort_name ::= sort_symbol sort
sort_symbol ::= ident
qualifier ::= ...
operator ::= \{ op | bop \} operator_symbol : [ arity ] -> coarity [ op_attrs ]
arity ::= sort_name ...
coarity ::= sort_name
op_attr ::= constr | associative | commutative | idempotent | \{ id : idr \} "(" term ")"
| strat: "(" natural ... ")" | prec: natural | l-assoc | r-assoc | coherent | demod
variable ::= var var_name : sort_name | vars var_name ... : sort_name
var_name ::= ident
axiom ::= equation | cequation | transition | ctransition | fol
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 "="
fol ::= axl \{ label \} term "="
label ::= ["ident ... "]:
macro ::= \#define term := term "="

1The nonterminal ident is for identifiers and will be defined in the section 1.3
2module_expr is defined in the section 1.3
3If optional \{ protecting | extending | including \} is omitted, it is defaulted to protecting
4comment is discussed in section 1.3
5There must not be any separators (see section 1.3) between ident and qualifier.
6operator_symbol is defined in section 1.4
7natural is a natural number written in ordinal arabic notation.
1.3 Module Expression

\[
\begin{align*}
\text{module_expr} & ::= \text{module_name} | \text{sum} | \text{rename} | \text{instantiation} | \text{"("} \text{module_expr}\text{"\")}\n\text{sum} & ::= \text{module_expr} \{ + \text{ module_expr} \} \ldots \\
\text{rename} & ::= \text{module_expr} \ast \text{"("} \text{rename_map}\text{\")"} \\
\text{instantiation} & ::= \text{module_expr} \text{"("} \text{ident[Qualifier]} \leq \text{aview}, \ldots \text{")"}\n\text{rename_map} & ::= \text{sort_map} \mid \text{op_map} \\
\text{sort_map} & ::= \text{\{ sort } \mid \text{hsort } \text{\} sort_name \rightarrow \text{ ident} \\
\text{op_map} & ::= \text{\{ op } \mid \text{bop } \text{\} op_name \rightarrow \text{ operator_symbol} \\
\text{op_name} & ::= \text{operator_symbol} \mid \text{"("} \text{operator_symbol}\text{")"qualifier} \\
\text{aview} & ::= \text{view_name} \mid \text{module_expr} \\
& \quad | \text{view to module_expr \"\"view_elt, \ldots \"\")"} \\
\text{view_name} & ::= \text{ident} \\
\text{view_elt} & ::= \text{sort_map} \mid \text{op_view} \mid \text{variable} \\
\text{op_view} & ::= \text{op_map} \mid \text{term} \rightarrow \text{term}
\end{align*}
\]

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:

\[ \text{sum} < \text{rename} < \text{instantiation} \]

1.4 View Declaration

\[
\begin{align*}
\text{view} ::= \text{view view_name from module_expr to module_expr} \\
& \quad \text{"\"view_elt, \ldots \"\")"}
\end{align*}
\]

1.5 Evaluation

\[
\begin{align*}
\text{eval} & ::= \{ \text{reduce} | \text{behavioural-reduce} | \text{execute} \} \text{ context term "\"} \\
\text{context} & ::= \text{in module_expr}:
\end{align*}
\]

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.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td>mod</td>
</tr>
<tr>
<td>module!</td>
<td>mod!</td>
</tr>
<tr>
<td>module*</td>
<td>mod*</td>
</tr>
</tbody>
</table>

Module Declaration

\[
\begin{align*}
\text{make} ::= \text{make module_name "("} \text{module_expr} \text{")"}
\end{align*}
\]

\text{make} is a short hand for declaring module of name module_name which imports module_expr with protecting mode.

\text{make FOO (BAR * \{sort Bar -> Foo\})}

is equivalent to

\text{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:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>protecting</td>
<td>pr</td>
</tr>
<tr>
<td>extending</td>
<td>ex</td>
</tr>
<tr>
<td>including</td>
<td>inc</td>
</tr>
<tr>
<td>using</td>
<td>us</td>
</tr>
</tbody>
</table>

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

\[ \text{ops } \text{operator}_1 \cdots \text{operator}_n : \text{arity} \to \text{coarity} \text{ op attrs} \]

is just equivalent to the following multiple operator declarations:

\[ \text{op } \text{operator}_1 : \text{arity} \to \text{coarity} \text{ op attrs} \]
\[ \vdots \]
\[ \text{op } \text{operator}_n : \text{arity} \to \text{coarity} \text{ op attrs} \]

bops is the counterpart of ops for behavioural operators.

\[ \text{bops } \text{operator}_1 \cdots : \text{arity} \to \text{coarity} \text{ op attrs} \]

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:

\[ \text{predicate ::= pred } \text{operator}_1 : \text{arity [ op attrs ]} \]

The form

\[ \text{pred } \text{operator}_1 : \text{arity op attrs} \]

is equivalent to:

\[ \text{op } \text{operator}_1 : \text{arity } \to \text{ Bool op attrs} \]

Operator Attributes  The following abbreviations are available:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>associative</td>
<td>assoc</td>
</tr>
<tr>
<td>commutative</td>
<td>comm</td>
</tr>
<tr>
<td>idempotent</td>
<td>idem</td>
</tr>
</tbody>
</table>

\[ ^8 \text{You cannot use sort_name of the same character sequence as that of any keywords, i.e., module, op, vars, etc. in arity.} \]
Axioms  For the keywords introducing axioms, the following abbreviations can be used:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceq</td>
<td>cq</td>
</tr>
<tr>
<td>trans</td>
<td>trns</td>
</tr>
<tr>
<td>btrans</td>
<td>btrns</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>red</td>
</tr>
<tr>
<td>bereduce</td>
<td>bred</td>
</tr>
<tr>
<td>execute</td>
<td>exec</td>
</tr>
</tbody>
</table>

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). 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.

Footnote: 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. ids 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.)

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 adjacent non-self-terminating tokens.

Comments also act as separators, but their apperance is limited to some specific places (see section 1).
Multiline comments  A multiple lines which starts with |#| and ends with |#| is treated as multilne comment.

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>This is an example of multiline comment.</td>
</tr>
<tr>
<td>Multiline comments are used for large text</td>
</tr>
<tr>
<td>descriptions of code or to comment out chunks of</td>
</tr>
<tr>
<td>code while developing your specification.</td>
</tr>
<tr>
<td>Multiline comments are ignored by the system.</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
</tr>
</tbody>
</table>