

# Introducing OBJ3's New Features

Timothy Winkler\*

August 22, 2001

**Abstract:** Some new features have been added to OBJ3 (as of the 2.0 release). The syntax and usage of these new features are briefly described in this document.

## 1 Introduction

This is a brief introduction to some of the newer features of OBJ3 added since Release 1.0. This introduction is written assuming that you have access to *Introducing OBJ3* by Goguen and Winkler [1]. Two major goals of the new features are (a) easy extension of modules and (b) the controlled application of rules. This last allows the use of OBJ3 as a basis for very flexible equational reasoning.

Descriptions of syntax here use the conventions of the *Introducing OBJ3* report. See also the beginning of the Syntax section, 3, where they are summarized.

## 2 Descriptions

The following sections give brief discussions of the syntax and usage of the new features. The syntax is also more formally presented in section 3.

### 2.1 Verbose mode

The option `verbose` controls how verbose various kinds of output will be. In verbose mode, detailed information will be given on sorts and the handling of automatically created `id` (identity) rules and conditions.

### 2.2 Treatment of Theories

Theories are now treated in nearly the same fashion as objects. It is possible to do reductions in a theory, though warnings will be issued for reductions using terms with free variables. See Section 2.23 for more information. The main remaining difference between objects and theories is that built-in equations are not allowed in theories. For more information on built-ins, see [2].

### 2.3 Including

There is a new mode of module incorporation called `including`.

```
including <ModExp> .  
inc <ModExp> .
```

The second form is an abbreviated version of the first. `Including` is like `using` in that no preservation of the structure of the included module is implied. On the other hand, in the implementation it is treated as incorporation without copying and so is similar to `protecting`. If a module is `included` twice in a given context, only one version of it is created and all references are all to the same shared instance.

---

\*This document was edited for clarity by Joseph Kiniry to be included with the 2.06 release of OBJ3 (June, 2000). Any errors herein are his alone. Please provide feedback to [obj3-feedback@kindsoftware.com](mailto:obj3-feedback@kindsoftware.com).

## 2.4 Principal Sort

The sort that should be the principal sort of a module can be explicitly specified by

```
principal-sort <Sort> .  
psort <Sort> .
```

The second form is an abbreviated version of the first. Note that this doesn't create a sort, it just specifies that an existing sort should be taken as the principal sort of the module being defined. Usually this is not needed, as the default choice of principal sort is correct. This operator is typically necessary if the principal sort is to come from a parameter theory. Recall that the notion of a principal sort plays an important role in default view computations, i.e. in abbreviated parameterized module instantiation a la `make`.

## 2.5 Vars-of and Show Vars

A new way of introducing variables into a module has been defined. You can now ask that new copies of all the variables of another object be introduced into the current object. These variables will maintain their name and sort.

```
vars-of [ <ModExp> ] .
```

Note: the set of variables considered to be in a module is not necessary cumulative. In particular, it may be somewhat smaller than you would expect.

Often it is the case that the module that introduces a sort also establishes a convention for naming variables of that sort, as well as introducing several variables of that sort. This new feature makes it easy to maintain the convention and reuse the variable names. This feature can also be used to make the variables of a re-opened module available for use (see later discussion). As usual, if the *<ModExp>* is omitted, the last module is used, but this is only really useful for such a re-opened module.

The variables of a module can be seen by using the command

```
show vars [ <ModExp> ] .
```

## 2.6 Let

A shorthand notation for defining a name for a term has been introduced.

```
let <Sym> = <Term> .  
let <Sym> : <Sort> = <Term> .
```

The name used is restricted to be a single simple symbol (such as “x”, “@2”, or “ThePoint”). The second form above is equivalent to

```
op <Sym> : -> <Sort> .  
eq <Sym> = <Term> .
```

In the first form, the sort of the operator is taken to be the sort of the term as discovered by parsing. When the defined symbol is used in expressions, it will be replaced by its definition, and so the symbol essentially stands for the value of the term. Note that, a consequence of this form is that each time the symbol is used, the term will be reduced once again.

There is a special case of let

```
let <Sym> [: <Sort> ] = .
```

This introduces a name for the current term which is last reduction result or the current state of the last `start`-ed term. See Section 2.25 for more information on `start`.

## 2.7 Show Principal Sort

The commands

```
show principal-sort [  $\langle ModExp \rangle$  ] .
```

and

```
show psort [  $\langle ModExp \rangle$  ] .
```

will display the principal sort of a given module, or the current module if none is specified.

## 2.8 Comments

If the first non-blank character after “\*\*\*” is a ‘(’, then the comment extends from that character up to the next balancing ‘)’. This block-comment form make it easy to comment out many lines at once. It is possible that comments written for the old convention may now get the wrong treatment. Examples of comments of this form are

```
*** (
  eq X * 0 = 0 .
)
*** (This is an interesting equation: ) eq X + X = X .
```

Note that the comment in the second case only extends over only part of a line. The equation is not part of the comment. This case in particular might cause trouble for existing comments. Standard style comments can be nested inside of multi-line comments, as long as a “\*\*\*” doesn’t accidentally appear within a line.<sup>1</sup>

## 2.9 The Last Module

The module name `THE-LAST-MODULE` will evaluate to the last module created or the current module being examined.

## 2.10 Openr, Open, and Close

It is possible to add terms to a module after it has been terminated by `endo` or `endth`.

If the module has been incorporated into some other module, either directly (e.g., `protecting mod`) or indirectly (by appearing in a module expression), then, after terms has added to the first module, the other module will no longer be valid and should no longer be used.

One particular goal of this new feature is to allow reductions in a partially defined object.

When it is desired to add more terms to a module, it is opened using

```
openr  $\langle ModExp \rangle$  .
```

or

```
openr .
```

The second form, as usual, opens the last module defined/used. Elements can be added to the open module using exactly the same syntax as is used to introduce them in the normal way in a module. All other commands (`in`, `set`, `show`, `select`, and `do`) continue work as usual. Normally a module that is opened should eventually be closed via

```
close
```

---

<sup>1</sup>This multi-line comment style has semantics similar to the `verbatim` environment in `LATEX`.

The system separately keeps track of a “last” modules and of an “open” module. Thus it is possible to show the module `INT` while the module `LIST` is open (making `INT` the “last” module), but still add elements to `LIST`. Additions of elements always go into the “open” module.

After a module is re-opened, the variables of that module are not available, but they can be made available by the previously discussed `vars-of` operator, specifically one of the form

```
vars-of .
```

There is an alternate version of `openr` called `open` which creates a hidden object (called “%”) and includes the given object. When the open object is closed in this case, the hidden object “%” is deleted. This allows the easy creation of an object that temporarily has more structure than the given object. The name `openr` was chosen to suggest “open retentive.” Note: If you `show` the open module, it will pretend to have the name of the underlying module with a marker (e.g. “\*\*\* open”) to remind you of this fact.

## 2.11 Select

A more natural way of selecting a module as the “last” module has been added. One may use

```
select [ <ModExp> ] .  
select open .
```

The second form makes the “open” module the “last” module (i.e., the default for `show` commands, etc.) “open” can be used as a short name for the “open” module in any of the `show` commands. The old command form `show select <ModExp>` has been retained.

## 2.12 Quietly Evaluate Lisp Forms

The command

```
eval-quiet <LISP>
```

can be used to evaluate a LISP form with the minimum of output during the processing of definitions. This can be abbreviated to `evq`.

## 2.13 Show rules

The commands for controlled application of rules require that you specify rules by number. The command

```
show rules [ <ModExp> ] .
```

shows a numbered list of rules of the module (defaulting to the “last” module) suitable for use in specifying a rule in an `apply` command. See the Section 2.26 for more information. The variant `show all rules .` will show all the rules in `verbose` and `all rules` modes.

There is a new option `all rules` which can be use to control the comprehensiveness of rule display with the `show` command. By default, rules not in the current module and in certain pre-defined modules are excluded. The command

```
set all rules on .
```

will make the set of rules more comprehensive.

## 2.14 Labeled Rules

A label can now be specified for a rule. The syntax for labels is:

```
[ <LabelList> ] <Rule>
```

A *<LabelList>* is a comma separated list of lists of identifiers. Identifiers must not contain a “.”, or begin with a digit.

The label need not immediately precede the rule. The form `[label]` can be thought of a setting the label for the next rule that is created. Example:

```
[def1] let x = 100 .
```

will work as expected. I.e., the label “def1” will be associated with the rule “x = 100” that is generated internally by the use of `let` operation.

Labels are shown when rules are displayed, and can likewise be used when rules are specified. Automatically generated rules (retractions, effects of parameterized instantiation, etc.) have automatically generated names.

## 2.15 Show a Rule

The command

```
show rule <RuleSpec> .
```

will show the selected rule. The variant `show all rule <RuleSpec> .` will show a specific rule in *verbose* mode.

## 2.16 Show Modules

The command

```
show modules .
```

shows a list of all currently defined modules.

If a module is redefined and is a simple-named module, then it may appear in the output from this command more than once.

## 2.17 Show Abbreviation for Module

One can see the abbreviation for a module’s name by using

```
show abbrev [ <ModExp> ] .
```

Abbreviations are *aliases* for modules that can be used in many top-level commands. They are of the form `M<number>`. These are just considered abbreviations, and are not really considered part of the syntax of the language and are thus implementation-dependent.

## 2.18 Showing Module in Detail

The command

```
show all [ <ModExp> ] .
```

will show a module in a more detailed form (more similar to original release of OBJ3). The default display has been changed to be somewhat closer to the form that of the definition of the module. The option `verbose` controls whether objects are displayed in detailed form by default.

## 2.19 Changes to the Standard Prelude

A new object IDENTICAL has been created that can be used in place of BOOL and provides two new operators === and /==. These check for syntactic equality of terms without evaluation or consideration of operator attributes.

The operator \_==\_ has been changed to be polymorphic. This can help with parsing problems in case a condition is of the form E == C, where C is an ambiguous constant (there is more than one constant with the same name) and E is has a well-defined unique sort.

The module THAT has been renamed to LAST-TERM, and the operator [that] has been renamed to [term].

The module RAT has been slightly changed so that the built-in constant values are printed like 1/2, not 1 / 2, and the same syntax (1/2) can be used for the input of these constants. (Previously, there was no syntax for the input of the constants.)

The object BUILT-IN has been modified to allow the creation of built-in subterms of RHSs of rules. The default syntax is “built-in: *<Lisp>*”, where the Lisp expression is represents a function that takes one argument, which is a substitution, and produces two values, a term which is the intended instantiation for this subterm, and a success indicator. In general, it will be necessary to deal with the incompatibility of the Sort Built-in with the sorts of other operators in the RHS. Here is a sketch of a use of this feature:

```
op r : Universal -> A .
var X : A .
eq f(X) = X + r(built-in: (lambda (u) (create-term u))) .
eq r(X) = X .
```

Note that “built-in:” is now a very special keyword, and cannot be used in any other context (this can be disabled by “ev (setq obj\_BUILT-IN\$keyword nil)”).

The object LISP has been added that provides a built-in Lisp sort. The default syntax is “lisp: *<Lisp>*”. This can be used to allow the use of string data with Lisp syntax for the strings. The keyword that introduces the data (above “lisp:”) can be changed to be some other symbol by **setq**-ing the variable obj\_LISP\$keyword to that other token (e.g. “string:”). Note that “lisp:” is now a very special keyword, and cannot be used in any other context (this can be disabled by “ev (setq obj\_LISP\$keyword nil)”).

## 2.20 Tracing of Rewriting

The options trace and trace whole are now independent and the format of the output has been made clearer in both cases. (It was the case that if trace was off, then the setting of trace whole was ignored.)

## 2.21 Files and Directories

The top-level command

```
cd <Directory>
```

changes the current directory for the running instance of OBJ to be the given directory or the current user’s home directory if the directory is omitted. The command cd can be used to change to one’s home directory.

The command cwd reports the current working directory.

The command

```
ls
```

lists the files in the current working directory.

If a file name begins with “ /”, then this will be expanded to the user’s home directory in most contexts.

(Note: these features are somewhat system dependent, and, because of portability problems, may not exist in all versions of OBJ.)

## 2.22 Command Line Arguments

In the AKCL version of OBJ3, there are some special command line arguments. These are pairs of these forms

```
-in <FileName>
-inq <FileName>
-evq <FileName>
```

The first two forms cause a file to be read in as OBJ3 starts up, either with a trace or quietly. The last form will quietly load a Lisp file on startup.

## 2.23 Variables in Reductions

Variables are now allowed (with a warning) in terms for reduction and many other contexts. The variables are treated as constants of appropriate sorts. This can be used to help find parsing problems since you can now provide subterms of a larger term to see if they parse and they may now contain variables.

## 2.24 Term and Show Term

The commands for the controlled reduction of a term operate by applying rules (or other actions) to a single term that is the focus of action. This single term will be called “term” and can be seen by using the command

```
show term .
```

(Initially it has a special undefined value.) This term is also modified by term reductions (command **red**) to be the final result of the reduction (this behavior is the same as in release 1.0). (This command existed in release 1.0, but it’s behavior is not exactly the same.)

## 2.25 Start

One can start working on the controlled reduction of a term by giving that term in a **start** command.

```
start <Term> .
```

The term given becomes the new focus of action and can be seen by using the **show term** command. (To reiterate: this focus is also set by the **red** command, and so is always the last reduction result or the current state of the last **start**-ed term.)

The **start** command can be used to examine the parse of a given term. After “term” has been set by **start** one can use **show term** to see its structure (in detail with **print with parens on**).

## 2.26 Apply

The `apply` command allows the selective application of rules (or other actions) at specified places in the current `term`. Furthermore, rules can be instantiated before they are applied. The syntax is rather complex: An `apply` command involves an *action*, an optional *instantiation*, a *range*, and a *selected subterm*. These elements will be discussed in the following subsections. A sketch of the syntax of the `apply` command is

```
apply action [ instantiation ] range selection
```

### 2.26.1 Action

An action is either a request to print the subterm, to reduce the subterm, or to apply a selected rule (possibly reversed) to a subterm.

```
print
reduction
red
<ModId>.<Int>
-<ModId>.<Int>
<ModId>.<Id>
-<ModId>.<Id>
```

The last four forms request the  $\langle Int \rangle$ th rule of module  $\langle ModId \rangle$  or that rule with the given `Id` as one of its labels (as shown by a `show rules` command), and that rule reversed. The  $\langle ModId \rangle$  must be a simple named module. This is a case where module name abbreviations are useful. If a label is used and multiple rules have that label, then the intent (this is not yet implemented) is to have the of rules be used in place of a single rule.

There are two special actions that allow the use of the implicit retract rules.

```
retr
-retr with sort <Sort>
```

The first rule applies the retract rules at the given point. These are rules of the form  $r:A>B(X:A) = X:A$ . The second form allows the introduction of a retract using the reversal of this rule; the `sort` given corresponds to  $B$  in  $r:A>B(X:A) = X:A$ .

### 2.26.2 Instantiation

When rules are reversed there may be variables in the new RHS that don't appear in the LHS. E.g.,

```
eq X * 0 = 0
```

reversed is

```
eq 0 = X * 0
```

In cases like this, it is logically necessary to specify a binding of the variable `X` in order to be able to apply the rule reversed. It is natural to allow this instantiation in all cases (even for non-reversed rules). Specifying an instantiation may make it possible to apply a rule without specifying a specific subterm (use `within` as the range, as discussed next).

An instantiation is specified by giving a substitution that specifies the bindings of some variables as a list of equations separated by commas (after the rule specification and delimited by `with`).

```
with <Var> = <Term> {, <Var> = <Term>}...
```

The variables bound must appear somewhere in the rule.

If some variable appears in the RHS, but not the LHS, and no binding is given, no warning is given. It is rather easy to notice that this has been done, since a variable will be introduced into the current `term`.

Instantiation is ignored if the action is to print or reduce.

### 2.26.3 Within or At

Reduction and printing are always for a whole subterm. Applying a rule can have one of two ranges either it is applied exactly “`at`” the selected subterm (see the next subsection) or it is applied anywhere “`within`” the selected subterm. In this latter case, if it is applied at a given point, it won’t be reapplied at that point or within the resulting subterm at that point.

### 2.26.4 Selection of Subterm

There are three basic kinds of selection: selection of an *occurrence*, *subsequence* (for associative operators), or *subset* (for associative commutative operators). There are also compositions of these basic selectors and in any case the selection process starts with the current `term`.

Subterms and argument positions are numbered from 1.

Selection of an occurrence looks like

$$\langle \langle Int \rangle \dots \rangle$$

The selection process proceeds from the starting term by successively passing to the argument positions specified by the successive integers. E.g., if the term is `(a + (c * 2))`, then the occurrence `(2 1)` selects the subterm `c`. The selector `()` simply selects the whole subterm (it is a selector, but it doesn’t cause a shift in focus).

Selection of a subsequence has two forms

$$\begin{aligned} & [ \langle Int \rangle \dots \langle Int \rangle ] \\ & [ \langle Int \rangle ] \end{aligned}$$

Spaces are required around the “`..`”. Selecting `[k]` is the same as selecting `[k .. k]`. This kind of selection is only appropriate for terms whose top operator is associative (or associative and commutative). For such operators, a tree of terms formed with that operator is naturally viewed as the sequence of the terms at the leaves of this tree. The form `[ <Int> ]` selections the *<Int>*th subterm of this logical sequence. (It doesn’t form a sequence of length one, which isn’t possible, in fact.) The form `[ <Int> .. <Int> ]` forces the restructuring of the term so that the specified range of terms of the logical sequence are a proper subterm of the whole term and then selects that term as the next current subterm. This means that a selection may change the exact structure of the term (so that a print request may affect the structure of the term).

Selection of a subset of the terms has the form

$$\{ \langle Int \rangle [ , \langle Int \rangle ] \dots \}$$

(Here the “`{}`” are not syntactic meta-notation, they just stand for corresponding characters.) No spaces are required within this notation. This kind of selection is only appropriate for terms with top operators that are associative and commutative. If an *<Int>* is repeated, these repetitions are ignored. This selector forces the given subset of the logical sequence (or more properly “bag”) of terms that are under the top operator to be a proper subterm and selects that term as the next current subterm. The order of the subterms in the “`{}`”s affects the order of appearance of these terms in the selected subterm. For example, in INT, if the current term is (when fully parenthesized) “`(1 * (2 * (3 * (4 * 5))))`”, the command

```
apply print at 1,3,5 .
```

is performed, then the resulting structure term is “((1 \* (3 \* 5)) \* (2 \* 4))”.

You can specify the top of “term” by either of the selectors

```
top
term
```

It only really makes sense to use these once and they can be omitted unless there is no other selector. In fact, one could also use the selector ().

### 2.26.5 Composition of Selectors

You can form a composition of selectors by separating them by `of`. For example,

```
{3,1,2} of [4] of (2 3 1)
[2 .. 5] of (1 1) of term
```

The interpretation of such a composition is like functional composition, the selection on the right is done first, then the middle one on the result of that selection, and then, finally, the one on the left. Note that this is the opposite of the order of interpretation of the elements of an occurrence (e.g., (2 1)).

### 2.26.6 Apply command and examples

The form of an apply command is `apply` followed (in order) by the action, possibly a substitution, `within` or `at`, and a composition of selectors. The schema for the apply command is

```
apply { reduction | red | print | retr |
      -retr with sort <Sort> |
      <RuleSpec> [ with <VarId> = <Term>{,
                  <VarId> = <Term>}... ] }
{ within | at }
<Selector> { of <Selector> }...
```

(Here “{}” are being used for syntactic grouping.) The resulting value of the current `term` is always printed after an apply command is performed. As stated before, when `try reduce conditions` is on, tests are forced to return `true` in rules that are explicitly applied.

Here are some examples.

```
apply G.1 at term .
apply -G.1 at term .
apply -G.2 with X = a at term .
apply print at term .
apply reduction at (2 1) .
apply G.1 at () .
apply X.3 at {2} .
apply X.3 at {3,1,2} .
apply G.2 at [2 .. 4] .
apply G.1 at [2] .
apply X.1 at {2,4} of [4] of (2 2) .
apply X.1 at {2,4} of [4 .. 4] of (2 2) of top .
```

The command `apply ?` . will display a summary of the usage of the apply command.

## 2.26.7 Conditional Rules

Allowing the controlled application of conditional rules requires, in general, that it be possible to shift the focus of reduction to the (instantiated) condition of a rule, thus allowing controlled application of rules to this condition. This is done by maintaining a stack of pending actions and pushing the application of a rule on the stack if its LHS matches, but it has a condition that needs to be carefully evaluated. When a condition reduces to “true” the delayed application of a rule is completed, and focus shifts back to the resulting term. If the condition instead reduces to “false”, then application of the rule is skipped, but you still shift focus back to the previous term.

In fact, it is possible to request that conditions of conditional equations be directly reduced. The command

```
set reduce conditions on .
```

requests this treatment. Naturally, the default behavior can then be reinstated by

```
set reduce conditions off .
```

(Either all nontrivial conditions are to be carefully evaluated, or none are.) One reason that one might prefer that the condition be directly evaluated is that, if the top operator of the LHS has special pattern matching attributes, then when the rule is applied all possible matches are tested against the condition until a successful case is found. On the other hand, with controlled application, only one match is attempted (an this is part of the reason that [2 .. 3], and the other forms, are needed).

Here is a small example. The object X has the definition

```
obj X is
  sort A .
  pr QID .
  subsort Id < A .
  op f : A -> A .
  var X : A .
  cq f(X) = f(f(X)) if (f(X) == 'a) .
  eq f('b) = 'a .
endo
```

Here is a sample output trace.

```
=====
start f ( 'b ) .
=====
apply X.1 at term .
shifting focus to condition
condition(1) Bool: f('b) == 'a
=====
apply X.2 within term .
condition(1) Bool: 'a == 'a
=====
apply red at term .
condition(1) Bool: true
condition is satisfied, applying rule
shifting focus back to previous context
result A: f(f('b))
```

Note that when actions are pending, “condition” is printed instead of “result” and the number of conditions being reduced (the number of pending actions) is printed in parentheses.

If you are evaluating a condition and want to force either success or failure you can use the following commands

```
start true .
start false .
```

For example, the above example could have continued from “apply X.1 at term” with

```
=====
start false .
condition(1) Bool: false
condition is not satisfied, rule not applied
shifting focus back to previous context
result A: f('b)
```

Thus, you can use this to abandon reductions that you no longer wish to perform. Use of “start true .” can easily produce incorrect reduction results, i.e. that do not follow by order-sorted equational deduction. If it is known that the condition being replaced is true, then this is not an issue. However, you cannot perform a controlled reduction in the middle of doing another one, and then continue the first reduction. (A new start causes the current state of the current term to be lost.)

If you want to see a description of all the pending actions you can use the command

```
show pending .
```

Which prints the details about the terms, rules, conditions, and replacements that are all currently pending. For example, the output of this command might look like

```
pending actions
1| in f('b) at top
  | rule cq f(X) = f(f(X)) if f(X) == 'a
  | condition f('b) == 'a replacement f(f('b))
2| in f('b) == 'a at f('b)
  | rule cq f(X) = f(f(X)) if f(X) == 'a
  | condition f('b) == 'a replacement f(f('b))
3| in f('b) == 'a at f('b)
  | rule cq f(X) = f(f(X)) if f(X) == 'a
  | condition f('b) == 'a replacement f(f('b))
```

If you use the range specification `within` and the rule is conditional, only one (at most) application of the rule will be reduced in a controlled way. You will be warned about this by a message like

```
applying rule only at first position found: f('b)
```

## 2.27 Identity Attribute Rule Completion Process

The handling of pattern matching for operators with identities requires a (partial) identity rule completion process which may result in some automatically created rules and may also result in special “id conditions” (that are normally not displayed). The partial completion and the id condition generation will be called **id processing**. The point is that rewriting modulo identity can often lead to non-termination problems. The id processing done in OBJ3 release 2.0 restricts the standard identity completion process to avoid simple cases of non-termination by adding id conditions to

rules (so that obviously problematic instances are disallowed) and also by discarding rule instances whose lefthand sides are variables (because their implementation as rules is problematic); in addition, rules subsumed by other rules are omitted for the sake of efficiency. Note: strategies of operators are not taken into account when testing for non-termination. It is possible that a rule will be considered as non-terminating, when non-termination is actually avoided because of the evaluation strategies. In `verbose` mode, many details of this process are displayed. When rules are displayed in a verbose way (either in `verbose` mode, or with a `show all` command), then the special id conditions will be displayed. When a module is processed, in `verbose` mode, some of the details of the completion process are shown, including rule instances that are generated and an indication of modifications to or additions of rules. The rules that have been automatically added by id processing will have automatically generated labels of the form “`compl(NAT)`”. An alternative approach to avoiding non-termination is to make the problematic rules conditional so as to prevent the undesired rewriting. The object `IDENTICAL`, provided in the standard prelude and used in this example, is `BOOL` with the addition of two operations `_===_` and `_=/=_` which test for syntactic equality and inequality, and are very useful hand-crafted identity conditions.

For example, if the following module is processed in verbose mode,

```
obj TST is
  sort A .
  ops c d e 0 1 : -> A .
  vars X Y : A .
  op _+_ : A A -> A [assoc comm id: 0] .
  eq X + Y = c .
jbo
```

the system will produce this output:

```
=====
obj TST
Performing id processing for rules
For rule: eq X + Y = c
  Generated valid rule instances:
  eq X + Y = c
  Generated invalid rule instances:
  eq Y = c
  eq X = c
  Modified rule: eq X + Y = c if not (Y === 0 or X === 0)
Done with id processing for rules
=====
```

In this example no new rule is generated, but the given rule had a special “id condition” added, which is not always displayed. A rule instance is invalid if the LHS is a variable (this is an implementation limitation), or if it would “obviously” cause non-termination, e.g. the LHS and RHS are the same term.

```
OBJ> show rule .1 .
rule 1 of the last module
  eq X + Y = c
OBJ> show all rule .1 .
rule 1 of the last module
  eq X + Y = c if not (Y === 0 or X === 0)
OBJ>
```

For a somewhat more complicated example consider:

```
obj TST is
  pr TRUTH-VALUE .
  sort A .
  op 0 : -> A .
  op _+_ : A A -> A [assoc id: 0] .
  op 1 : -> A .
  op *_ : A A -> A [assoc id: 1] .
  op f : A -> A .
  ops a b c d e f : -> A .
  var X Y : A .
  eq (X * Y) + f(X * Y) = f(X) .
endo
```

The verbose output would be:

```
=====
obj TST
Performing id processing for rules
For rule: eq (X * Y) + f(X * Y) = f(X)
  Generated valid rule instances:
  eq (X * Y) + f(X * Y) = f(X)
  eq X + f(X) = f(X)
  eq Y + f(Y) = f(1)
  eq f(0) = f(1)
  Generated invalid rule instances:
  eq f(0) = f(0)
  Added rule: [compl16] eq f(0) = f(1)
  Added rule: [compl17] eq X + f(X) = f(X) if not X === 0
  Modified rule: eq (X * Y) + f(X * Y) = f(X) if not (X === 0 and Y ===
  1)
Done with id processing for rules
=====
```

In this case, the rule `compl16` was added because the top operator is `f` not `_+_`. In the other case, rule `compl17`, the LHS of the rule is a strict generalization of the original LHS, i.e. the original rule LHS cannot match the new one (`X + f(X)`). It might make sense to delete the original rule, but this is never done. Note also that confluence is assumed, as always, and in this context this means confluence for all rule instances and for all ways of matching. Here this means that it is valid not to add the rule `eq Y + f(Y) = f(1)`. (Of course, this is a very contrived example and the original system was not confluent.)

## 2.28 Help commands

The command `?` (note: no “.”) produces the following output

```
Top-level definitional forms include: obj, theory, view, make
The top level commands include:
q; quit --- to exit from OBJ3
show .... . --- for further help: show ? .
set .... . --- for further help: set ? .
do .... . --- for further help: do ? .
```

```

apply ..... --- for further help: apply ? .
other commands:
  in <filename>
  red <term> .
  select <module-expression> .
  cd <directory>; ls; pwd
  start <term> .; show term .
  open [<module-expression>] .; openr [<module-expression>] .; close
  ev <lisp>; evq <lisp>

```

The “;”s are only used to separate alternatives.

## 2.29 Problems

It is possible that the rules (as seen in a `show rules` command) will be reordered in curious ways when modules are constructed. There may be odd repetitions of rules in certain cases.

## 3 Syntax

This section gives the syntax of the new features of for OBJ3, using the an extended BNF notation as an extension of the presentation of the syntax of OBJ3 in [1]. The symbols { and } are used as meta-parentheses; the symbol | is used to separate alternatives; [ ] pairs enclose optional syntax; ... indicates 0 or more repetitions of preceding unit; and "x" denotes x literally. As an application of this notation, A{A}... is an idiom used for a non-empty list of As separated by commas. .... is used to mark the omission of other alternatives which are described in [1]. Finally, --- indicates comments in the syntactic description (as opposed to comments in OBJ3 code).

```
--- modules ---
```

```

<ModElt> ::= ..... |
  { using | extending | protecting |
    including } <ModExp> . |
  principal-sort <Sort> . |
  let <Sym> [ : <Sort> ] = <Term> . |
  let <Sym> [ : <Sort> ] = . |
  vars-of [ <ModExp> ] .

```

```
<Sym> --- any operator syntax symbol (blank delimited)
```

```
--- top-level ---
```

```

<OBJ-input> ::= { ..... |
  <RuleLabel>
  openr [ <ModExp> ] . |
  open [ <ModExp> ] . |
  close |
  start <Term> . |
  <Apply>
  }...

```

```
<RuleLabel> ::= [ <Id>...{,<Id>...}... ]
```

```

⟨Apply⟩ ::=
  apply { reduction | red | print | retr |
        -retr with sort ⟨Sort⟩ |
        ⟨RuleSpec⟩ [ with ⟨VarId⟩ = ⟨Term⟩{,
                    ⟨VarId⟩ = ⟨Term⟩}... ] }
  { within | at }
  ⟨Selector⟩ { of ⟨Selector⟩ }...

```

```

⟨RuleSpec⟩ ::= [-][⟨ModId⟩].⟨RuleId⟩

```

```

⟨RuleId⟩ ::= ⟨Natural Number⟩ | ⟨Id⟩

```

```

⟨Selector⟩ ::= term | top |
  (⟨Int⟩...) |
  "[" ⟨Int⟩ [ .. ⟨Int⟩ ] "]" |
  "{"⟨Int⟩{,⟨Int⟩}..."}"
  --- note that "()" is a valid selector

```

```

⟨Commands⟩ --- show, set, do, select
in particular:

```

```

  select [ ⟨ModExp⟩ ] .
  show [all] rules [ ⟨ModExp⟩ ] .
  show [all] rule ⟨RuleSpec⟩ .
  show vars .
  show term .
  select open .
  set reduce conditions ⟨On/Off⟩ .
  set all rules ⟨On/Off⟩
  show [all] rule ⟨RuleSpec⟩
  eval-quiet ⟨LISP⟩
  show abbrev [ ⟨ModExp⟩ ] .
  show modules .
  show all [⟨ModExp⟩] .
  set verbose ⟨On/Off⟩ .
  show pending .

```

```

open --- can use this to refer to the open module in show commands

```

```

⟨On/Off⟩ ::= on | off

```

```

⟨Comment⟩ ::= *** ⟨Rest-of-line⟩ | ***> ⟨Rest-of-line⟩ |
  *** ( ⟨Text-with-balanced-parentheses⟩ )

```

```

--- equivalent forms ---

```

```

inc = including
evq = eval-quiet
psort = principal-sort

```

## 4 Last words

I would like to acknowledge many useful suggestions from José Meseguer that helped improve these notes. Good Luck!!!

## References

- [1] Joseph Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab, August 1988. Revised version to appear with additional authors José Meseguer, Kokichi Futatsugi and Jean-Pierre Jouannaud, in *Applications of Algebraic Specification using OBJ*, edited by Joseph Goguen, Derek Coleman and Robin Gallimore, Cambridge, 1992.
- [2] Timothy Winkler and José Meseguer. OBJ3's Built-ins. SRI International, Menlo Park, CA 94025. Included with OBJ3 releases 2.05 and newer.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Descriptions</b>	<b>1</b>
2.1	Verbose mode . . . . .	1
2.2	Treatment of Theories . . . . .	1
2.3	Including . . . . .	1
2.4	Principal Sort . . . . .	2
2.5	Vars-of and Show Vars . . . . .	2
2.6	Let . . . . .	2
2.7	Show Principal Sort . . . . .	3
2.8	Comments . . . . .	3
2.9	The Last Module . . . . .	3
2.10	Openr, Open, and Close . . . . .	3
2.11	Select . . . . .	4
2.12	Quietly Evaluate Lisp Forms . . . . .	4
2.13	Show rules . . . . .	4
2.14	Labeled Rules . . . . .	5
2.15	Show a Rule . . . . .	5
2.16	Show Modules . . . . .	5
2.17	Show Abbreviation for Module . . . . .	5
2.18	Showing Module in Detail . . . . .	5
2.19	Changes to the Standard Prelude . . . . .	6
2.20	Tracing of Rewriting . . . . .	6
2.21	Files and Directories . . . . .	6
2.22	Command Line Arguments . . . . .	7
2.23	Variables in Reductions . . . . .	7
2.24	Term and Show Term . . . . .	7
2.25	Start . . . . .	7
2.26	Apply . . . . .	8
2.26.1	Action . . . . .	8
2.26.2	Instantiation . . . . .	8
2.26.3	Within or At . . . . .	9
2.26.4	Selection of Subterm . . . . .	9
2.26.5	Composition of Selectors . . . . .	10
2.26.6	Apply command and examples . . . . .	10
2.26.7	Conditional Rules . . . . .	11
2.27	Identity Attribute Rule Completion Process . . . . .	12
2.28	Help commands . . . . .	14
2.29	Problems . . . . .	15
<b>3</b>	<b>Syntax</b>	<b>15</b>
<b>4</b>	<b>Last words</b>	<b>17</b>