Bourbaki Proof Checker

Juha Arpiainen

February 15, 2006

Contents

1 Introduction

Bourbaki is a program for writing and verifying formal mathematical proofs. It is inspired mainly by Norman Megill's *Metamath* and the related programs Ghilbert and Mmj2, but aims on providing more powerful syntax and tools for automated proof writing.

Nicolas Bourbaki is the pseudonym of a group of mathematicians, who have written a rigorous multi-volume treatise of Mathematics emphasizing structures. My long-term goal with Bourbaki is an online hyperlinked, computer-verified encyclopedia of Mathematics, perhaps in conjunction with the Hyperreal Dictionary of Mathematics project.

Bourbaki is implemented as an embedded language on top of ANSI Common Lisp. This means most of Lisp features are available for the user, in particular it is possible to write Lisp functions that construct proofs (See section 9).

The code of Bourbaki is roughly divided in two levels. The *symbol level* code implements a simple proof system. The *reference level* code provides a hierarchical namespace system to organize the information. The names refer to the principal data types used on the respective levels. There are also I/O methods for converting between various data formats formats.

Sections 2 to 4 of this document describe the symbol level part of Bourbaki. The reference level code is described in sections 5 to 7. More advanced, programmable features are described in sections 8 and 9. Finally, section 10 deals with the various methods of importing and exporting Bourbaki data to and from other formats.

1.1 Quick start

```
$ tar xvf bourbaki.tar.gz
$ cd bourbaki
 clisp
> (load "init")
> (in-package bourbaki-user)
> (verify !!prop)
> (print-theorem !/prop!id)
```

1.2 Note on definitions

We refer to the Common Lisp specification for definitions of terms such as *form* (anything that can be evaluated by Lisp) and *string*.

New terms will be defined in BNF syntax. For example,

```
strings := string | ( string* )
```

defines *strings* to be either a string or a list of strings.

2 Symbols and Formulae

Mathematical expressions are treated as trees of symbols in Bourbaki. Thus the expression $2(a + b)$ is stored internally as $(*_s\ (2_s)\ (+_s\ (a_s)\ (b_s)))$, where the $*_s, 2_s, \ldots$ are Lisp objects corresponding to the multiplication operator, the number 2, etc.

One usually calls the function `symtree-parse` to create such expressions. Such calls are abbreviated with the `[ ]` syntax: evaluating the form `[* 2 + a b]` would construct the above expression. In this document we will sometimes use the `[ ]` notation to denote the symtree resulting from such a call, although this is technically incorrect.

Note that we are using the so-called *Polish prefix notation*, where parentheses can be omitted if each operator takes a fixed number of arguments (which is usually the case in Bourbaki).

2.1 Symbol kinds

Each expression in Bourbaki has a *syntactic type* or *kind*. For example, expressions of first order logic are *variables*, *terms* or *well formed formulae* (*wffs*).

Such types are declared with the macro `symkind`, for example, `(symkind "wff")`.

A syntactic type may be declared a subtype of another type by giving the `:super` keyword to `symkind`. For example, the line `(symkind "var" :super term)` says that every variable is a term.

The function `(bsubtypep x y)` returns true if `x` and `y` are the same type or if `x` is a subtype of `y`. An expression e can be substituted for a variable v if `(bsubtypep (type of e) (type of v))` is true. If this is the case we say e is of *correct type* for v.

2.2 Primitive symbols

Primitive symbols are defined with macro `prim`:

```
prim-form := (prim symkind name varspec body)
name      := string
varspec   := ( { symkind strings }* )
body      := form*
```

For example, the definition

```
(prim wff "=" (term "x" term "y"))
```

states that `[= x y]` is a wff for any two terms `x` and `y`. Here `x` and `y` are the *variables* of the symbol `=`. The number of variables is called the *arity* of the symbol.

Several variables of the same kind can be declared by putting their names in a list:

```
(prim wff "=" (term ("x" "y")))
```

would have worked. Bourbaki defines a reader macro for '`![`' for abbreviating such lists:

```
(prim wff "=" (term ![x y]))
```

is equivalent to both of the above definitions.

The names of symbols and their variables are case-sensitive. The symbols are stored in a namespace separate from Lisp symbols. A reference to the symbol can be obtained using the '`!`' reader macro: `!=` returns a reference to the equality symbol just defined. Inside brackets the use of '`!`' is optional: `[= a b]` or `[!= !a !b]`.

It is possible to redefine a symbol: the new definition replaces the original. Symtrees may continue to point to the original symbol, however. The old and the new definition are never the same symbol (in the sense of `eq`), even if they have the same arity and variables of the same types.

2.3 Symbol trees

A *symbol tree*, or *symtree* for short, is a list $(op\ arg_1\ arg_2\ \ldots\ arg_n)$ such that *op* is a Bourbaki symbol and each arg_i is a symtree. *op* is called the *operator* of the tree and the arg_i are *arguments*. The leaves of the tree are symbols with no arguments ($n = 0$ is allowed in the definition).

More generally, the *op* may be any Bourbaki context (see section 5), in particular a theorem. We also call such lists symtrees.

A *symtree pattern* is defined similarly, but the arguments may also be integers or Lisp symbols.

The *type* of a symtree (*op* arg_1 ... arg_n) is the symbol kind of *op*. The tree is *well formed* if the arity of *op* is n and each arg_i is well formed and of correct type for the corresponding variable of *op*. In this document the term *wff* is used for any well-formed symtree.

2.4 Symtree functions

The well-formedness of a symtree can be tested with the function `wffp`.

Two symtrees can be compared with `wff-equal`. `(wff-equal x y)` returns true if `x` and `y` have isomorphic tree structure and the corresponding symbols are `eq`.

`wff-type` returns the type of a symtree.

A common operation is to simultaneously substitute all occurrences of a number of symbols in a symtree with some other symtrees. This operation is performed by `replace-vars`, a function taking the symtree and a list of pairs (sym_i . $subst_i$) called the *substitution map* or *smap* in the code.

The result of substituting sym_i with $subst_i$ in the symtree tr is denoted by $tr(sym_1/subst_1, \ldots, sym_n/subst_n)$ or $tr(subst_1, \ldots, subst_n)$ if the sym_i are clear from context.

`(print-symtree tree stream)` prints the tree using the bracket notation. The default value of `stream` is `*standard-output*`.

3 Theorems and Proofs

Theorems and *axioms* in Bourbaki have zero or more *variables*, *hypotheses* and *assertions*. In addition a theorem must have a *proof*. The variables are Bourbaki symbols of arity zero; the hypotheses and assertions are wffs, usually containing the variables. In the simple case the proof is a sequence of theorem references justifying the assertions.

The theorem is treated as a transformation rule: when its hypotheses are satisfied for some symtrees substituted for its variables, its (correspondingly substituted) assertions can be used to prove other theorems.

Theorems are defined using the macros `th`, `hypo`, `ass`, and `proof`:

```
theorem-form     := (th name varspec body)
hypothesis-form  := (hypo symtree*)
assertion-form   := (ass symtree*)
proof-form       := (proof proof-line*)
proof-line       := symtree
```

Axioms are defined like theorems, but using `ax` instead of `th`:

```
axiom-form := (ax name varspec body)
```

An example from propositional calculus follows:

```
;; the implication symbol
(prim wff "->" (wff ![x y]))

;; the axioms
(ax "ax1" (wff ![A B])
  (ass [-> A -> B A]))
(ax "ax2" (wff ![A B C])
  (ass [-> -> A -> B C -> -> A B -> A C]))

;; modus ponens
(ax "ax-mp" (wff ![A B])
  (hypo [A] [-> A B])
  (ass [B]))

;; identity law for '->'
;; see below for interpretation of the proof
(th "id" (wff "A")
  (ass [-> A A])
  (proof
    [ax1 A [-> A A]]
    [ax2 A [-> A A] A]
    [ax-mp [-> A -> -> A A A]
           [-> -> A -> A A -> A A]]
    [ax1 A A]
    [ax-mp [-> A -> A A] [-> A A]]))
```

As seen from this this example, the `hypo` (and `ass`) lines are optional. Multiple hypothesis and assertions lines can be used:

```
(hypo [A])
```

`(hypo [-> A B])`

works as well. In contrast, each `proof` replaces the previous one. Hypothesis and axiom lines can be freely mixed, and the order usually does not matter (but see section 9). The proof should be after all hypotheses and assertions. Bourbaki allows a `proof` expression for an axiom, although such proof won't be used anywhere.

Theorems and axioms live in the same namespace with symbols. The rules concerning redefinition of symbols apply also to theorems. A theorem's variables, hypotheses, assertions and proof can be printed with `print-theorem`.

3.1 Verifying proofs

The function `verify` checks the correctness of a theorem or axiom *thr*. The (somewhat simplified) verification algorithm follows:

1. Check that the hypotheses and assertion of *thr* are well-formed.
2. If *thr* is an axiom, we are done. Otherwise initialize list L with the hypotheses of *thr*.
3. For each proof line `[`*ref* $subst_1$ ... $subst_n$`]`,
   - Verify ref if it is not already verified.
   - Check that the number of variables of ref is equal to the number n of substitutions, and that each $subst_i$ is well-formed and of correct type.
   - For each hypothesis h of ref, check that h`(`$var_1/subst_1, \ldots, var_n/subst_n$`)` is in L, where $var_1, \ldots, var_n$ are the variables of ref.
   - For each assertion a of ref, insert a`(`$subst_1, \ldots, subst_n$`)` into L.
4. Check that each assertion of *thr* is in L.

The verifier detects *vicious circles* (theorems referring to themselves or cycles of theorem references).

3.2 Distinct variables

Sometimes the simple substitution rule of Bourbaki causes problems. Consider for example the theorem $\forall x \exists y (x \neq y)$ of predicate calculus (which is true assuming there are at least two objects). Simple substitution of y for x would result in the incorrect formula $y \neq y$.

In the above theorem x and y should be marked *distinct* using the macro `dist`:

```
(th "exists2" (var ![x y])
  (dist (!x !y))
  (ass [∀ x ∃ y ≠ x y])
  (proof ...))
```

When two variables `x` and `y` are marked distict in a theorem, substitution of a for `x` and b for `y` is valid only if

- no variable occurs in both a and b, and
- each variable occurring in a is marked distinct from each variable in b.

Lists of more than two variables can be given to `dist`. All variables in the list are made pairwise distinct. The general form of `dist` is

```
distinct-form := (dist condition*)
condition     := ( variable* )
```

The reason distinct variable conditions are necessary is that the variables of Bourbaki are *metavariables* rather than individual variables of the object language; see the Metamath book for more information.

3.3 Local variables

Sometimes it is necessary to use additional variables in a proof, not occurring in hypotheses or assertions. Such 'dummy' or 'local' variables can be defined with macro `loc`, for example `(loc var "z")`.

Local variables are treated as distinct from each other and the 'normal' variables.

4 Definitions

Writing everything in terms of primitive symbols gets tedious quickly. New symbols can be defined in terms of previous ones with the `def` macro.

```
definition-form := (def def-name symkind sym-name
                        sym-vars other-vars rhs body)
def-name        := string
sym-name        := string
sym-vars        := varspec
other-vars      := varspec
rhs             := symtree
```

This behaves as if a primitive symbol was defined with

```
(prim symkind sym-name sym-vars)
```

together with an axiom

```
(ax def-name (sym-vars other-vars)
  (ass [eq sym-name sym-vars rhs])) .
```

Here ***eq*** is the *equality operator* for the symbol kind of the defined symbol. For example, the equality operator for wffs is the biconditional '↔'. The equality operator can be set with

```
(setf (symkind-eq-op wff) (mkcontext !↔)) .
```

Note that in this case the biconditional must be a primitive symbol, `(prim wff "↔"(wff ![ϕ ψ]))`, it cannot be used to define itself.

4.1 Soundness of definitions

A definition is *sound* if

- each formula containing the defined symbol can be proved (using the definition) to be equivalent to a formula without the symbol, and
- for each theorem whose hypotheses and assertions do not contain the defined symbol, there is a proof that does not use the definition.

These conditions state that definition does not add anything new to the language: it is just an abbreviation for a pattern of primitive symbols.

Definitions of the form `[eq sym` $var_1 \ldots var_n$ `rhs]` are provably sound if the equality operator is an *equivalence relation* satisfying the *substitution laws*, that is, the following formulae are theorems:

- `[eq a a]` (Reflexivity)
- `[→ eqa b eq b a]` (Symmetry)
- `[→ ∧ eqa b eqb c eqa c]` (Transitivity)
- `[→ eqa b eq'` ϕ`(a)` ϕ`(b)]` for any formula ϕ. Here `eq'` is the equality operator for the type of ϕ; ϕ`(a)` denotes the formula where `a` is properly substituted for `x` in ϕ`(x)`.

Ensuring these conditions hold is a responsibility of the user.

4.2 Bound Variables

The right-hand-side of a definition may contain primitive symbols, already defined symbols, variables of the defined symbol, and additional *bound variables*, the `other-vars` of the definition. A symbol can be made to bind a variable by giving an additional `:bound` argument in the `varspec`. The universal quantifier should properly be specified

```
(prim wff "∀" ((:bound var) "x" wff "φ")) .
```

The `other-vars` of a definition are implicitly marked bound and distinct from each other and the `sym-vars`.

All occurrences of a variable `x` in a symtree `[op` $subst_1 \ldots subst_n$`]` are bound by `op` if `x` occurs in a substitution $subst_i$ such that the corresponding variable of `op` is marked bound.

A definition `[eq sym` $var_1 \ldots var_n$ `rhs]` is valid only if all potential occurrences of each variable `x` marked bound are bound in `rhs`. The validity is verified by the function `verify`, otherwise definitions are treated like axioms by the verifier.

4.3 Sample Definitions

```
;; Conjunction
(def "df-and" wff "∧" (wff ![φ ψ]) ()
```

```
  [¬ → φ ¬ ψ])

;; Existential quantification
(def "df-ex" wff "∃" ((:bound var) "x" wff "φ")
  [¬ ∀ x ¬ φ])

;; Proper substitution. Theorem "sb7" in set.mm
;; [sb a x φ] is the wff where a is properly
;; substituted for x in φ.
(def "df-subst" wff "sb" (term "a" var "x" wff "φ") (var "y")
  [∃ y ∧ = y a
     [∃ x ∧ = x y φ]])
```

In the third example, `x` cannot be marked bound: if the term substituted for `a` contains free occurrences of the variable substituted for `x`, those occurrences are free in the `rhs`. This explains the term *potential occurrences* used above. If `x` were distinct from `a`, the definition would be valid.

5 The Context System

For larger projects it gets inconvenient to have all theorems and symbols in the same namespace. Bourbaki provides a hierarchy of namespaces, called *contexts*, to place related symbols and theorems in. At the top of the hierarchy there is a *root context*, the value of `*root-context*`. Changing the value of `*root-context*` allows several (possibly unrelated) systems to be loaded at the same time. It is also possible to have contexts outside any hierarchy.

Contexts are defined with macros `module` and `context`. `module` creates a *top-level context*, that is, a context directly below the root of the hierarchy; `context` is used to create nested subcontexts.

A sample hierarchy follows. This will be referred to in following examples.

```
(module "logic"
  (context "propositional"
    (prim wff "→" (wff ![φ ψ])))
  (context "predicate")
    (prim wff "∀" (var "x" wff "φ")))
(module "set-theory"
  (context "zfc"
    (context "axioms"
      (ax "union" ...)
```

```
    (ax "choice" ...))))
```

At any time one of the contexts, the value of `*current-context*`, is selected. New symbols and theorems are inserted into `*current-context*` by default. Macros such as `context` and `module` bind `*current-context*` to the context being defined. The macro `in-context` explicitly binds `*current-context*`.

Symbols and theorems are retrieved from the context hierarchy with the function `seek-sym`, usually abbreviated with the reader macro `!`. The following example illustrates the use of `in-context` and `seek-sym`.

```
;; !/ specifies an absolute reference,
;; with respect to *root-context*
;; Same as (print-theorem
;;           (seek-sym :abs "set-theory" "zfc"
;;                          "axioms" "choice"))
(print-theorem !/set-theory!zfc!axioms!choice)

(in-context !/set-theory!zfc
  ;; Single ! specifies a reference
  ;; relative to *current-context*
  ;; Same as (print-theorem (seek-sym :rel "axioms" "union"))
  (print-theorem !axioms!union)

  (in-context !axioms
    ;; Add an axiom
    (ax "pairing" ...))

  (th "foo" (wff "φ")
    ;; ! can be used within []
    (ass [!/logic!propositional!→ φ φ])))
```

Symbols and theorems are actually contexts is Bourbaki, and may contain subcontexts. For example, the variables of a theorem *th* are symbols that are subcontexts of *th*. The words *symbol* and *context* are used somewhat interchangeably in this document.

5.1 Meta information

Each context contains a table of meta information indexed by Lisp symbols. Some fields are to be filled by the user, others by Bourbaki or other programs.

Meta information can be accessed with `(gethash` *key*`(context-meta` *context*`))`
The macro `meta` sets meta fields of the current context:

meta-form := (meta (*key* . *content*)*)
key := *symbol*
content := *form*

The following fields are used by the current version of Bourbaki. They are all symbols in the `keyword` package.

:assume, :bound, :comment, :def, :defs-used, :depends-on, :description, :full-name, :html-fn, :html-proof-fn, :html-sym, :index, :original, :parent, :proof-length, :used-by

5.2 Nested variables and hypotheses

In mathematical theories it is common to have theorems with common variables and hypotheses. For example, theorems in the theory of topological spaces would all have a variable `X` and a hypothesis `[topo-space X]`. Such common variables and hypotheses can be moved to the parent context:

```
(module "topology"
  (def wff "topo-space" (set "X") ...)

  (theory "topo" (set "X")
    (hypo [topo-space X])

    ;; closure of A in the topology of X
    (def set "closure" (set "A") ...)

    (th "some-theorem" (set "Y") ...)))
```

The macro `theory` is just a version of `context` that allows specification of variables. When used from outside `topo` this behaves as the following:

```
(in-context !/topology
  (context "topo"
    (def set "closure" (set "X" set "A") ...)
    (th "some-theorem" (set "X" set "Y")
      (hypo [topo-space X])
      ...)))
```

Within `topo` the space `X` is used by default:

```
(in-context !/topology!topo
  (print-symtree [closure ∅]))
⇒[closure X ∅]
```

This is a case of *importing* symbols, as defined in the next section. When variables from multiple contexts are nested, the variables of the innermost context are on the *rightmost*.

5.3 Context functions

New contexts are created with function `create-context`:

```
(create-context :name string :parent context
                :type symkind :class symbol)
```

`parent` defaults to `*current-context*`, a `nil` parent creates a context outside the hierarchy. `class` should be one of the following symbols:

:axiom, :definition, :theorem, :prim, :sym, :arg, :loc, :context, :module

`create-context` does not yet insert the new context to parent's symbol table, see `insert-sym` in the next section.

`mkrootcontext` creates a context without a parent, suitable value for `*root-context*`. `flush` sets `*root-context*` to a new root context, usually sending the previous hierarchy to the garbage collector.

All symbols and theorems in Bourbaki are structures of type `context`:

```
(defstruct context
  name
  type
  class
  (meta (make-hash-table 'eq))

  vars
  hypo
  assert
  distinct
  proof

  (syms (make-hash-table 'equal))
  imports
  exports)
```

The `syms`, `imports` and `exports` are describend in the next section.

There is some redundancy. The `proof`, for example, does not make sense for a primitive symbol. Future versions of Bourbaki might take a more object-orinted approach.

6 Symbol References

A *symbol reference* or *symref* is a function taking symtrees as arguments and returning a symtree. They are used to represent argument transformations. For example, the call `(in-context !/topology!topo !closure)` returns a function taking one argument. When applied to `A` the function returns `[closure X A]`.

6.1 Terminology

The symbol references are structures of type `symref`:

```
(defstruct symref
  target
  args-needed
  fn)
```

`target` is the referred context, or `nil` if the symref doesn't refer to a specific context. `args-needed` is an integer n describing the function `fn`.

If $n \geq 0$, `fn` is a function taking a *fixed number* $m \geq n$ of arguments. The symref is *exact* if $m = n$.

If $n < 0$, `fn` is a function taking a *variable number* of arguments, at least $|n|-1$. In the rest of this section we assume that $n \geq 0$, symrefs with variable number of arguments are treated in section ¡Metasyms¿.

A symref is *composable* if `args-needed` ≥ 0 and the `target` is not `nil`. The function `(mkcontext x)` returns `x` itself if it is a context, and the target of `x` if `x` is a composable symref. Many of the Bourbaki functions taking context arguments also accept composable symrefs, calling `mkcontext`. `verify` and `print-theorem` are examples of these.

6.2 Composing References

The fundamental operation of symrefs is composition. Given an exact, composable symref `f` of n arguments and a symref `g` of m arguments, `(compose-ref f g)`

is an exact symref of $n + m$ arguments. It is composable if and only if `g` is.

When the function of `(compose-ref f g)` is called with arguments $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m$, the first n arguments are given to `f`. `f` should return a list $c_1, c_2, \ldots, c_k$ of symtrees, where $m + k$ is the actual number of arguments g expects. The result of calling `(compose-ref f g)` is then the result of calling `g` with $c_1, \ldots, c_k, b_1, \ldots, b_m$.

Corresponding to a theorem

```
(th "example" (type1 "var1" ... typeN "varN")
    ...)
```

the `syms` hashtable of the parent context contains a symref with `args-needed = N` under the key `"example"`. The symref function just passes on its arguments without modification. The same applies also to contexts of other type.

A reference `!/foo!bar!example` composes these references while going down the context hierarchy, resulting in a symref with `example`'s arity.

Actually the order of arguments is opposite to the one described here: `f` takes the *last* n arguments of `(compose-ref f g)`. This makes the code simpler and more efficient.

6.3 The Parser

The function `symtree-parse` takes arguments of various types and attempts to construct a symtree from them. If the first argument to `symtree-parse` is a list, it is assumed to be an already parsed symtree and returned as is. If the first argument is an integer or a Lisp symbol, a one-element list containing it is returned. This allows `symtree-parse` to be used to create symtree patterns. Otherwise the first argument must be a symref or a context.

If the first argument *op* is a context with n variables, `symtree-parse` is called recursively to parse n subexpressions. The symtree `[`*op* $sub_1, \ldots, sub_n$`]` is then returned.

If *op* is a symref with `args-needed` $= n \geq 0$, the function of *op* is applied to the next n subepressions in *reversed order* (see the note at the end of 6.2). *op* must therefore be an exact symref.

If $n < 0$, subepressions are parsed until all the arguments are used up. *op* is then applied to the subexpressions in reversed order.

Interpretation of the function call depends on the `target` of *op*. If `target` is `nil`, the result is assumed to be a valid symtree and is returned as is. Otherwise the result is assumed to be a list of arguments to `target` in reversed

order, and the symtree `(`*target* `. (reverse` *result*`))` is returned.

The reader macro for '`[`' is an abbreviation for `symtree-parse`. Whitespace-separated tokens are read until a '`]`' is seen. Tokens beginning with one of the characters '`[`', '`!`', '`"`', '`(`', or '`,`' receive a special treatment. Otherwise the token is a string naming a symbol in `*current-context*`. Thus `[+ x y]` is an abbreviation for

```
(symtree-parse (seek-sym :rel "+")
               (seek-sym :rel "x")
               (seek-sym :rel "y"))
```

The special characters are treated as follows:

- `[` begins a nested symtree. This is necessary when dealing with symrefs taking variable number of arguments. It may be used to parenthesize the expression for clarity.
- `!` begins a call to `seek-sym` read with the usual syntax of `!`.
- `"` begins a string read with the Lisp reader and given to `seek-sym`. It may be used to escape the special characters in a symbol's name, for example, `[+ "[foo␣bar]" 3]`.
- When a '`,`' is encountered, the Lisp reader is used to read a form. The form is then evaluated and given to `symtree-parse`. This works like with backquote. Example:

  ```
  (let ((x [+ a b]) (y [- a b]))
    [* ,x ,y])
  ```

  produces the same symtree as `[* + a b - a b]`. `[-> ,1 ,2]` constructs a symtree pattern.
- `(`... is simply an abbreviation for `,(`....

6.4 Importing and Exporting

It gets tedious to write references such as `!/logic!propositional!`→ or `!/topology!topo!closure` all the time. Bourbaki allows the relevant contexts to be *imported*:

```
(module "test"
  (import !/logic!propositional)
  (import !/topology!topo)

  (print-symtree [→ x y])
  (print-symtree [closure A B]))

;; error: the import is only visible within "test"
(print-symtree [closure A B])
```

More generally, default values can be given to some of the variables of the imported context by using `import` with a symtree pattern:

```
(theory "continous" (set ![X Y])
  ;; f: X -> Y is continuous
  (def "df-cont" wff "cont" (set "f") ...))

(context "calculus"
  ;; Specialize to continuous real-valued functions
  (import [!/continuous ,0 IR])

  (print-symtree [cont A f]))
⇒[cont A IR f]
```

Import only allows references to be abbreviated within a context. To provide an everywhere accessible version of df.cont that defaults to IR one must use `export`:

```
(context "test1"
  (export [!/continuous R R]))
(context "test2"
  (import [!/continuous R R]))

(print-symtree [!test1!cont f])
⇒[cont R R f]

(print-symtree [!test2!cont f])
⇒Error: symbol "cont" not found
```

The Bourbaki import and export are analoguous to the Lisp functions with the same names; both deal with visibility of symbols. There are differ-

ences, however: Bourbaki import and export deal with whole contexts instead of individual symbols. The symbols and theorems defined in a context are also exported by default. The require and provide described in section 7 are something closer to Ghilbert's import/export.

6.5 Seeking symbols

6.6 Aliases

Sometimes it is necessary to export only a few symbols instead of the whole context. In such cases one can use `alias` macro.

`pattern` constructs such a symref without inserting it to current-context

7 Modules and Files

It is common to have theorems of similar form, but different symbols in their assertions. For example, the inclusion relation $a \subseteq b$ and the less-than-or-equal relation $\langle x, y\rangle \in \leq$ of real numbers are both partial orders (here the relation $\leq$ is interpreted as a set of ordered pairs of real numbers as usual).

Bourbaki allows writing "templates" for such theorems and re-using them in different contexts. In the above situation we could write

```
;;; File order-ax.lisp
;;; Axioms for partial orders

(module "order-ax"
  (import [!!logic-ax])
  (symkind "obj")
  (prim wff "≤" (obj ![a b]))

  ;; Define the associated strict order
  (def "df-less" wff "<" (obj ![a b])
    [∧ ≤ab ≠ab])

  ;; Antisymmetric law
  (ax "antisym" (obj ![a b])
    (ass [→ ∧ ≤ab ≤ba = ab])))
```

```
  ;; Transitive law
  (ax "tr" (obj ![a b c])
    (ass [→ ∧ ≤ ab ≤ bc ≤ ac])))

;;; File order.lisp
;;; Theorems for partial orders

(module "order"
  (import [!!logic])
  (import [!!order-ax])

  ;; Strict order is transitive
  (ax "less-tr" (obj ![a b c])
    (ass [→ ∧ < ab < bc < ac])
    (proof ...))
  ...)

set.lisp
calculus.lisp
```

Here the line `(import [!!order-ax])` tells Bourbaki to import the symbols from the top-level context `order-ax`, loading the file `order-ax.lisp` if the context was not found. The line `(provide ... (export [!!order]))` binds the name `order-ax` and the symbol kind `obj` to `provide-order-ax` and `set`, respectively. The file `order.lisp` is then loaded with these bindings and the resulting context is exported.

The end result is that `!/set!less-tr` is a theorem with assertion `[→ ∧ ⊂ a b ⊂ b c ⊂ a c]` and `!/calculus!less-tr` one with `[→ ∧ < a b < b c < a c]`.