Universidade do Minho

Escola de Engenharia

Departamento de Informática

Largo do Paço - 4709 BRAGA Codex- PORTUGAL

Tel +351-253604470- Fax +351-253612954

CAMILA: A Reference Manual

Luís Soares Barbosa
José João Almeida

DI-CAM-95:11:1:v98
Technical Report
October 1998

Camila: A Reference Manual

Luís Soares Barbosa
José João Almeida

lsb@di.uminho.pt
jj@di.uminho.pt

Departamento de Informática
Universidade do Minho
Largo do Paço
4709 BRAGA Codex
PORTUGAL

Tel +351-253604470
Fax +351-253612954

DI-CAM-95:11:1:v98
Technical Report

October 1998

Contents

WARNING

The present Reference Manual for CAMILA documents a system's version previous to CAMILA 2.00. This means there are a number of features supported either by the kernel prototyping language or the development system but not documented here. The most relevant concern:

- Type constructors and associated canonical functions (expressing functorial or universal properties). The material is covered in [BA99], Chapter 2.
- Available library of CAMILA models.
- Supporting tools for CAMILA, namely `pretty`, `interface`, `cam2man` and `camtex`. Some are briefly mentioned in [BA99]. Such different tools have been integrated in a (simple) development toolkit for CAMILA — `camtlk` — included in the distribution.

1 Introduction

1.1 Camila

From school physics we got used to a basic problem solving strategy: *create a mathematical model, reason on it, calculate a solution.* The Camila approach [ABNO97a, ABNO97b, ABBN98] is an attempt to make such a strategy available at the software engineering level. Based on a notion of *formal software component* it encompasses a set-theoretic notation, a prototyping environment, fully connectable to external applications and equipped with communication facilities, and an inequational refinement *calculus*.

The project affiliates itself, but is not restricted to, to the research in exploring Functional Programming as a *rapid prototyping* environment for formal software models, whose origin can be traced back to P. Henderson's me too [Hen84].

In the way, some new theoretical and technological results — namely a component classification and reification calculus and a notion of connectable high-level prototyping environment — were achieved and incorporated. At the same time educational projects and non trivial case-studies were carried on in industrial contexts and this shaped the original project as a software development platform, while retaining initial simplicity and stressing adaptability to specific application areas.

A Camila software component is described in the centenary notation of *naive* set theory. Furthermore, the kernel of the prototyping environment may be seen as a *calculator* for (constructive) set theory, in the sense that the basic constructions of the (cartesian closed) category of sets are used as type constructors in the language, which, in their turn, are acted by groups of (standard) associated functions. For a detailed account of the Camila approach to software design the interested reader may refer to [BA99]. Further information may be found on the Camila Www-page at `camila.di.uminho.pt`, from which a mailing list can be subscribed.

Camila is named after a Portuguese 19$^{\text{th}}$-century novelist — Camilo Castelo-Branco (1825 - 1890) — whose immense and heterogeneous writings, deeply rooted in his own time experiences and controversies, mirrors a passionate and difficult life.

1.2 The Manual

This report is a preliminary version of a Reference Manual for the Camila prototyping language, describing in detail the set constructors and primitive functions available. An introduction to the prototyping environment is also included.

The first subsection of Section 2 describes the set constructors which form the basic data models in the Camila prototyping language. For each of them a short description, the corresponding meaning in set theory and the associated xmetoo S-expression type is shown. Next the Camila primitive operators are presented and classified with respect to the data models involved and their general purpose. Each entry in the Manual collects the operator designation, alternative designations (if any), a short description, the xmetoo correspondent expression, an example and, when applicable, a semantic description in set theoretic notation. The syntax rule is given in the form of a grammatical production where non-terminal symbols are written between angle brackets. For easy reading, Non-terminal symbols identifiers were given the same designation of the corresponding xmetoo types. The only exception is the symbol $\langle S \rangle$ (from S-expression) which stands for the super-set of all the others non-terminals referred in the text. The (meta-)symbol $\equiv$, used in some examples, stands for expression equivalence.

Finally, Section 3 is a quick introduction to the prototyping environment.

2 Reference Manual

2.1 Basic Data Models

`NIL`

Description	The (isomorphism class of the) singleton set represented by {`NIL`}.
Xmetoo	`NIL`
Semantics	**1**, i.e. the terminal object in the category of sets.

`BOOL`

Description	The (isomorphism class of the) two-element set, referred to as the set of boolean values and represented by {`true,false`}.
Xmetoo	`BOOL`
Semantics	**2**

`L1:X1 L2:X2 ... Ln:Xn`   `L1:X1 * L2:X2 * ... * Ln:Xn`

Description: Cartesian Product of sets `X1` to `Xn`. The identifiers `L1` to `Ln` are (optional) labels denoting the projection morphisms associated with the product construction. When processing the type declaration `X = L1:X1 L2:X2 ... Ln:Xn`, Camila automatically adds to the prototype code the following structural functions

`X` : `X1` × `X2` × ... × `Xn` ⟶ `X`	*constructor*
`L1` : `X` ⟶ `X1`	*selector*
`L2` : `X` ⟶ `X2`	*selector*
...	...
`Ln` : `X` ⟶ `Xn`	*selector*

Xmetoo: `TUP`

Semantics: $X_1 \times X_2 \times ... \times X_n = \{(x_1, x_2, ..., x_n) | \ \forall_{i \in \{1,...,n\}} \, . \, x_i \in X_i\}$ with L_1 to L_n standing for the projections $\pi_1 : X_1 \times X_2 \times ... \times X_n \longrightarrow X_1$ to $\pi_n : X_1 \times X_2 \times ... \times X_n \longrightarrow X_n$.

`X1 | X2 | ... | Xn`

DESCRIPTION Coproduct — or disjoint union — of sets `X1` to `Xn`. When processing the type declaration `X = X1 | X2 | ... | Xn`, CAMILA automatically adds to the prototype code the following origin-identifier predicates

`is-X1` : `X` $\longrightarrow$ `BOOL`
`is-X2` : `X` $\longrightarrow$ `BOOL`
...
`is-Xn` : `X` $\longrightarrow$ `BOOL`

The notation `[X]` is used as a shorthand for `X|NIL`

XMETOO `ALT`

SEMANTICS $X_1 + X_2 + ... + X_n \;=\; (\{1\} \times X_1) \cup (\{2\} \times X_2) \cup ... (\{n\} \times X_n)$ where each origin-identifier predicate $is{-}X_j$ is defined, resorting to the canonical injection morphism $i_{X_j} : X_j \longrightarrow X_1 + X_2 + ... + X_n$, as $is{-}X_j(x) \;=\; \exists_{a \in X_j} \,.\, i_{X_j}(a) = x$. The semantics of `[X]` is obviously $X + \mathbf{1}$, where $\mathbf{1}$ stands for the singleton set.

`X-set`

DESCRIPTION (Finite) Sets of `X`

XMETOO `SET`

SEMANTICS 2^X

`X-seq (X-list)`

DESCRIPTION (Finite) Sequences of `X`

XMETOO `LIST`

SEMANTICS X^*

`X -> Y`

DESCRIPTION (Finite) Partial Functions from `X` to `Y`

XMETOO `FF`

SEMANTICS $X \rightharpoonup Y$

`X <-> Y`

DESCRIPTION (Finite) Binary Relations in the cartesian product of `X` and `Y`

XMETOO `REL`

SEMANTICS $R \subseteq X \times Y$

STR

Description	The set of strings, where a string is any sequence of characters started and ended by ", with a maximum of 100 characters. It may include special characters like \e (escape), \n (newline), \r (return), \tab (tabulation) and, in general, *ddd* (character whose ascii code is the octal value *ddd*).
Xmetoo	`STR`
Semantics	$Char^*$

INT

Description	The set of integer numbers, represented as no empty sequences of digits, with an optional prefix signal (`+` or `-`). The subset of $\mathbb{Z}$ representable by INT is the one assigned to the `int` type on the C language.
Xmetoo	`INT`
Semantics	$\mathbb{Z}$

SYM

Description	The set of formal symbols used as identifiers in any context and represented as no empty sequences of digits, letters, `$`, `_` and `!`, whose first character is not a digit. Evaluation of a value of type `SYM` always returns the value linked to it in the current prototyping environment.
Xmetoo	`SYM`
Semantics	A set of formal symbols

Environmental Types

Camila also provides some other primitive types which do not bear a direct mathematical correspondence but are inherent to its programming environment. They are listed in the following table.

`FPTR`	File pointers
`CHAN`	Communication channels
`SUBR`	Primitive functions
`FUN`	User defined functions
`TYPE`	Types

2.2 Finite Subsets

{}

Syntax Rule	⟨*SET*⟩ ::= {}
Description	{}
	▶ Empty set
	Constant standing for the empty set.
Xmetoo	`(empty)`
Example	`{4,5,7} * {}` ≡ `{}`
Semantics	$\emptyset$, i.e. the initial object in the category of sets.

`_U_` [`union(_,_)`]

Syntax Rule	⟨*SET*⟩ ::= ⟨*SET*⟩ `U` ⟨*SET*⟩
Description	`s U r`
	▶ Set union
	The set formed by union of sets `s` and `r`.
Xmetoo	`(union s)`
Example	`{4,5,7} U {x | x <- {7,2,9} : x >= 5}` ≡ `{4,5,7,9}`
Semantics	$s \cup r$

`UNION(_)`

Syntax Rule	⟨*SET*⟩ ::= `UNION(`⟨*SET*⟩ `)`
Description	`UNION(s)`
	▶ Distributed set union
	The set formed by union of all sets in set `s`.
Xmetoo	`(UNION s)`
Example	`UNION({{4,5,7},{12,5},{7,2,9}})` ≡ `{4,5,7,12,2,9}`
Semantics	$\bigcup_{x \in s} x$

`_*_` [`intersection(_,_)`]

Syntax Rule	⟨*SET*⟩ ::= ⟨*SET*⟩ `*` ⟨*SET*⟩
Description	`s * r`
	▶ Set intersection
	The set formed by intersection of sets `s` and `r`.
Xmetoo	`(intersection s r)`
Example	`{4,5,7} * {x | x <- {7,2,9} : x >= 5}` ≡ `{7}`
Semantics	$s \cap r$

`_-_` [`difference(_,_)`]

SYNTAX RULE	⟨*SET*⟩ ::= ⟨*SET*⟩ `-` ⟨*SET*⟩
DESCRIPTION	`s - r` ▶ Set difference The set formed by removing every element of set `r` from set `s`.
XMETOO	`(difference s r)`
EXAMPLE	`{4,5,7} - {x	x <- {7,2,9} : x >= 5}` ≡ `{4,5}`
SEMANTICS	$s - r$

`{_,_,...,_}`

SYNTAX RULE	⟨*SET*⟩ ::= `{` ⟨*S*⟩ [(`,` ⟨*S*⟩)*] `}`
DESCRIPTION	`{x1, x2, ..., xn}` ▶ Set enumeration The set consisting of elements `x1`, `x2`, ..., `xn`.
XMETOO	`(makeset x1 x2 ... xn)`
EXAMPLE	`{2,5,1}`
SEMANTICS	$\{x_1, x_2, ..., x_n\}$

`{_|_<-_,...:_}`

SYNTAX RULE	⟨*SET*⟩ ::= `{` ⟨*S*⟩ `|` ⟨*SYM*⟩ `<-` (⟨*SET*⟩ | ⟨*REL*⟩ | ⟨*LIST*⟩) [`,` ...] [`:` ⟨*BOOL*⟩] `}`
DESCRIPTION	`{e	x <- s : p}` ▶ Set comprehension The set resulting from applying expression `e`, on variable `x` to those elements of `s` verifying predicate `p`
XMETOO	`(set e (from x s p) ...)`
EXAMPLES	`{x .+ y	x <- {7,2,9}, y <- {2,5,1} : x <= 7}` ≡ `{9,12,8,4,7,3}` `{x U {# x}	x <- <{3,4},{9,1,2,5},{3,4},{1}> }` ≡ `{ {3,4,2},{9,1,2,5,4},{1} }`
SEMANTICS	$\{e(x_1, ..., x_n) \mid x_1 \in s_1, ..., x_n \in s_n \ \wedge\ p(x_1, ..., x_n)\}$.

`inseg(_)`

SYNTAX RULE	⟨*SET*⟩ ::= `inseg(` ⟨*INT*⟩ `)`
DESCRIPTION	`inseg(n)` ▶ Initial segment The set consisting of elements `1` up to `n`.
XMETOO	`(inseg n)`
EXAMPLE	`inseg(4)` ≡ `{1,2,3,4}`
SEMANTICS	$\overline{n}$

`_ == {}` [isempty(_)]

Syntax Rule	$\langle BOOL \rangle$ ::= $\langle SET \rangle$ `== {}`
Description	`s == {}` ▶ Empty set test True if `s` is the empty set.
Xmetoo	`(isempty s)`
Example	`{'peter,'paul,'mary} == {}` $\equiv$ `false`
Semantics	$x = \emptyset$

`_in_` [member(_,_)]

Syntax Rule	$\langle BOOL \rangle$ ::= $\langle S \rangle$ `in` $\langle SET \rangle$
Description	`x in s` ▶ Set membership True if `x` belongs to set `s`.
Xmetoo	`(member x s)`
Example	`'mary in {'peter,'paul,'mary}` $\equiv$ `true`
Semantics	$x \in s$

`_notin_`

Syntax Rule	$\langle BOOL \rangle$ ::= $\langle S \rangle$ `in` $\langle SET \rangle$
Description	`x notin s` ▶ Negative set membership True if `x` does not belong to set `s`.
Xmetoo	`(not (member x s))`
Example	`'mary notin {'peter,'paul,'mary}` $\equiv$ `false`
Semantics	$x \notin s$

`subset(_,_)`

Syntax Rule	$\langle BOOL \rangle$ ::= `subset(`$\langle SET \rangle$`,` $\langle SET \rangle$ `)`
Description	`subset(r,s)` ▶ Set inclusion True if `r` is a subset of set `s`.
Xmetoo	`(subset x s)`
Example	`subset({'mary}, {'peter,'paul,'mary})` $\equiv$ `true`
Semantics	$x \subseteq s$

`choice(_)`

Syntax Rule	$\langle S \rangle$ ::= `choice(`$\langle SET \rangle$`)`
Description	`choice(s)`
	▶ Non deterministic choice
	Chooses non deterministically an element from set `s`.
Xmetoo	`(choice s)`
Example	`choice({'peter,'paul,'mary})` $\equiv$ `'mary`
Semantics	∇s

`the(_)`

Syntax Rule	$\langle S \rangle$ ::= `the(`$\langle SET \rangle$`)`
Description	`the(s)`
	▶ Deterministic choice
	Chooses the (unique) element of set `s`.
Xmetoo	`(the s)`
Examples	`the({'mary})` $\equiv$ `'mary`
	`the({'peter,'paul,'mary})` $\equiv$ not defined
Semantics	the s

`# _` [`card(_)`]

Syntax Rule	$\langle INT \rangle$ ::= `#` $\langle SET \rangle$
Description	`# s`
	▶ Cardinal
	The cardinal number of set `s`
Xmetoo	`(card s)`
Example	`# {x	x <- {7,2,9} : x >= 5}` $\equiv$ `2`
Semantics	$\#s$

`_ -set`

Syntax Rule	$\langle FUN \rangle$ ::= $\langle FUN \rangle$`-set`
Description	`f-set`
	▶ Functorial extension of a function
	The function that applies `f` to each element of the (set) argument.
Xmetoo	-
Example	`succ-set({1,2,3})` $\equiv$ `{2,3,4}`
Semantics	$\mathbf{2}^f$

`_-red(_,_)` [`reduce(_,_,_)`]

Syntax Rule	$\langle S\rangle$ ::= $\langle SYM\rangle$`-red(` $\langle S\rangle$ `,` $\langle SET\rangle$`)`
Description	`o-red(e,s)` ▶ Distributed form of binary operation Applies to set `s` the iterated form of the associative binary function `o` whose unit element is `e`.
Xmetoo	`(reduce e o s)`
Examples	`.+-red(0,{1,2,3})` $\equiv$ `6` `U-red({},{{2,3},{2,4,7},{5}})` $\equiv$ `{2,3,4,7,5}`
Semantics	For the above examples the corresponding set expressions are $\Sigma_{n\in\{1,2,3\}}\ n$ and $\bigcup_{x\in\{\{2,3\},\{2,4,7\},\{5\}\}} x$.

2.3 Finite Functions

`[]`

Syntax Rule	$\langle FF \rangle$ ::= `[]`
Description	`[]` ▶ Empty mapping Constant standing for the empty mapping.
Xmetoo	`nil`
Example	`dom([])` ≡ `{}`
Semantics	$[\,]$

`_ == []` [`isempty(_)`]

Syntax Rule	$\langle BOOL \rangle$ ::= $\langle FF \rangle$ `==` `{}`
Description	`s == []` ▶ Empty mapping test True if `s` is the empty mapping.
Xmetoo	`(isempty m)`
Example	`[x -> #x	x <- {{"ann","peter"},{"ann"}} : "mary" in x] == []` ≡ `true`
Semantics	$x = [\,]$

`_[_]` [`ap(_,_)`]

Syntax Rule	$\langle S \rangle$ ::= $\langle FF \rangle$ `[` $\langle S \rangle$ `]`
Description	`m[x]` ▶ Application The value associated to element `x` in (the no empty mapping) `m`; `NIL` if $x \notin \mathrm{dom}\ m$.
Xmetoo	`(ap m x)`
Example	`[n -> factorial(n)	n <- {1,2,3,4,5,6,7,8,9}][4]` ≡ `24`
Semantics	$m[x]$

`dom(_)`

Syntax Rule	$\langle SET \rangle$ ::= `dom(`$\langle FF \rangle$`)`
Description	`dom(m)` ▶ Domain The domain of the mapping `m`.
Xmetoo	`(dom m)`
Example	`dom([x -> #x | x <- {{"ann","peter"},{"ann"}}])` ≡ `{{"ann","peter"},{"ann"}}`
Semantics	$\mathrm{dom}\ m$

`ran(_)`

SYNTAX RULE	⟨*SET*⟩ ::= `ran(`⟨*FF*⟩`)`
DESCRIPTION	`ran(m)` ▶ Co-Domain The co-domain of the mapping `m`.
XMETOO	`(ran m)`
EXAMPLE	`ran([x -> #x | x <- {{"ann","peter"},{"ann"}}])` ≡ `{2,1}`
SEMANTICS	rng m

`_ @ _` [`ds(_,_)`]

SYNTAX RULE	⟨*FF*⟩ ::= ⟨*FF*⟩ `@` ⟨*SET*⟩
DESCRIPTION	`m @ s` ▶ Domain subtraction The mapping obtained from `m` by removing the set `s` from the domain of `m`.
XMETOO	`(ds m s)`
EXAMPLE	`["ann"->12, "peter"->8, "mary"->14] @ {"ann","mary"}` ≡ `["peter"->8]`
SEMANTICS	$[x \mapsto m(x) \mid x \in (\text{dom } m - s)]$

`_ / _` [`dr(_,_)`]

SYNTAX RULE	⟨*FF*⟩ ::= ⟨*FF*⟩ `/` ⟨*SET*⟩
DESCRIPTION	`m / s` ▶ Domain restriction The mapping obtained from `m` by restricting the domain of `m` to the set `s`.
XMETOO	`(dr m s)`
EXAMPLE	`["ann"->9, "sue"->8, "mary"->14] / {"ann","mary"}` ≡ `["ann"->9, "mary"->14]`
SEMANTICS	$[x \mapsto m(x) \mid x \in (\text{dom } m \cap s)]$

`_ + _` [`plus(_,_)`]

SYNTAX RULE	⟨*FF*⟩ ::= ⟨*FF*⟩ `+` ⟨*FF*⟩
DESCRIPTION	`f+g` ▶ Overwrite The mapping obtained from `f` by replacing every element `i->f[i]` by `i->g[i]`, for all `i` in the domain of `g`.
XMETOO	`(plus f g)`
EXAMPLE	`[1->12, 2->8, 3->4] + [2->5, 4->4, 1->8]` ≡ `[1->8, 2->5, 3->4, 4->4`
SEMANTICS	$f \dagger g$

[_->_,_->_...,_->_]

SYNTAX RULE	$\langle FF \rangle$ ::= `[` $\langle S \rangle$ `->` $\langle S \rangle$ [(`,` $\langle S \rangle$ `->` $\langle S \rangle$)+] `]`
DESCRIPTION	`[d1->x1, d2->x2, ..., dn->xn]` ▶ Finite function enumeration The mapping consisting of pairs `d1->x1`, `d2->x2`, ..., `dn->xn`.
XMETOO	`(makeff (d1 x1) (d2 x2) ... (dn xn))`
EXAMPLE	`["ann"->12, "peter"->8, "mary"->14]`
SEMANTICS	$[d_1 \mapsto x_1, d_2 \mapsto x_2, ..., d_n \mapsto x_n]$

[_->_|_<-_:_]

SYNTAX RULE	$\langle FF \rangle$ ::= `[` $\langle S \rangle$ `->` $\langle S \rangle$ `|` $\langle SYM \rangle$ `<-` $\langle SET \rangle$ [`,` ($\langle SYM \rangle$ `<-` $\langle SET \rangle$)+] [`:` $\langle BOOL \rangle$] `]`
DESCRIPTION	`[ed -> er | x <- s : p]` ▶ Finite function comprehension The mapping resulting from applying expressions `ed` and `er` to those elements of `s` verifying predicate `p`. The expressions `ed`, `er` and `p` are, in general, dependent on the variable `x`. Notice that when forming pairs by expressions on more than one variable some pairs may be overwritten to guarantee that the functionality condition remains valid. For example `[x -> x .+ y | x<-{1,2,3}, y <-{5,6}]` evaluates for `[1->6,2->7,3->8]`.
XMETOO	`(ff (pair ed er) (from x s p))`
EXAMPLES	`[x -> # elems(x) | x <- {<0,0,0>,<1,1,1,0,1,0>,<0,0,0,1>,<1,1,1,1>}]` $\equiv$ `[<0,0,0> -> 1, <1,1,1,0,1,0> -> 2, <0,0,0,1> ->2, <1,1,1,1> -> 1]`
SEMANTICS	$[ed(x_1, ..., x_n) \mapsto er(x_1, ..., x_n) \| \, x_1 \in s_1, ..., x_n \in s_n \wedge p(x_1, ..., x_n)]$.

_ ->*

SYNTAX RULE	$\langle FUN \rangle$ ::= $\langle FUN \rangle$`->*`
DESCRIPTION	`f->*` ▶ Functorial extension of an injective function over the domain of a mapping The function that applies `f` to each element in the domain of the argument.
XMETOO	-
EXAMPLE	`succ->*([1 -> "a", 2 -> "b"])` $\equiv$ `[2 -> "a", 3 -> "b"]`
SEMANTICS	$f \rightharpoonup Id$

`*-> _`

SYNTAX RULE	$\langle FUN \rangle$ ::= `*->` $\langle FUN \rangle$
DESCRIPTION	`*->f` ▶ Functorial extension of a function over the codomain of a mapping The function that applies `f` to each element in the codomain of the argument.
XMETOO	-
EXAMPLE	`(*->ascii)([1 -> "a", 2 -> "b"])` $\equiv$ `[2 -> 97, 3 -> 98]`
SEMANTICS	$Id \rightharpoonup f$

2.4 Finite Sequences

<>

SYNTAX RULE $\langle \mathit{LIST} \rangle$ ::= <>
DESCRIPTION <>
▶ Empty sequence
Constant standing for the empty sequence.
XMETOO nil
EXAMPLE length(<>) $\equiv$ 0
SEMANTICS $<>$

_ ^ _ [append(_,_)]

SYNTAX RULE $\langle \mathit{LIST} \rangle$ ::= $\langle \mathit{LIST} \rangle$ ^ $\langle \mathit{LIST} \rangle$
DESCRIPTION s ^ r
▶ Sequence concatenation
The sequence formed by juxtaposing sequence r after sequence s.
XMETOO (append s r)
EXAMPLE <"ann","sue","bill"> ^ <"mary","ann"> $\equiv$ <"ann","sue","bill","mary","ann">
SEMANTICS $s \frown r$

CONC(_)

SYNTAX RULE $\langle \mathit{LIST} \rangle$ ::= CONC($\langle \mathit{LIST} \rangle$)
DESCRIPTION CONC(s)
▶ Distributed sequence concatenation
The sequence formed by concatenating all sequences in sequence s.
XMETOO (CONC s)
EXAMPLE CONC(<<4,5,7>,<12,5>,<7,2,9>>) $\equiv$ <4,5,7,12,5,7,2,9>
SEMANTICS $s_1 \frown s_2 \frown ... \frown s_n$ for $s = <s_1, s_2, ..., s_n>$

<_:_ > [cons(_,_)]

SYNTAX RULE $\langle \mathit{LIST} \rangle$ ::= <$\langle \mathit{S} \rangle$: $\langle \mathit{LIST} \rangle$>
DESCRIPTION <x:s>
▶ Appending
The sequence formed by appending element x to sequence s.
XMETOO (cons x s)
EXAMPLE <"ann":<"peter","bill">> $\equiv$ <"ann","peter","bill">
SEMANTICS $<x> \frown s$

`<_,_,...,_>`

Syntax Rule ⟨*LIST*⟩ ::= `<` ⟨*S*⟩ [(`,` ⟨*S*⟩)+] `>`

Description `<x1, x2, ..., xn>`

▶ Sequence enumeration

The sequence consisting of elements `x1`, `x2`, ..., `xn`.

Xmetoo `(makeseq x1 x2 ...  xn)`

Example `<2,5,1,3,5>`

Semantics $<x_1, x_2, ..., x_n>$

`<_,_>` [`pair(_,_)`]

Syntax Rule ⟨*LIST*⟩ ::= `<` ⟨*S*⟩ `,` ⟨*S*⟩`>`

Description `<x1, x2>`

▶ Pairing

The ordered pair consisting of elements `x1` and `x2` represented as a sequence of two elements.

Xmetoo `(pair x1 x2)`

Example `<5,5>` ≡ `<5> ^ <5>`

Semantics $<x_1, x_2>$

`<_|_<-_,...:_>`

Syntax Rule ⟨*LIST*⟩ ::= `<` ⟨*S*⟩ `|` ⟨*SYM*⟩ `<-` ⟨*SET*⟩ [(`,` ⟨*SYM*⟩ `<-` ⟨*SET*⟩)+] [`:` ⟨*BOOL*⟩] `>`

Description `<e | x <- s :  p>`

▶ Sequence comprehension

The sequence resulting from applying expression `e`, on variable `x`, to those elements of set `s` which verify predicate `p`

Xmetoo `(seq e (from x s p) ...)`

Example `< # x | x <- {{3,4},{9,1,2,5},{3,4},{1}} >` ≡ `< 2,4,2,1 >`

Semantics $< e(x_1, ..., x_n) | \; x_1 \in s_1, ..., x_n \in s_n \; \wedge p(x) >$

`reverse(_)`

Syntax Rule ⟨*LIST*⟩ ::= `reverse(`⟨*LIST*⟩`)`

Description `reverse(s)`

▶ Reversed sequence

The sequence obtained from `s` by reversing the positional order of its elements.

Xmetoo `(reverse s)`

Example `reverse(reverse(<"ann","peter","bill">))` ≡ `<"ann","peter","bill">`

Semantics $\mathrm{reverse}(\mathrm{tl}\ s) \frown <\mathrm{hd}\ s>$

`hd(_)`

Syntax Rule	$\langle S\rangle$ ::= `hd(`$\langle LIST\rangle$`)`
Description	`hd(s)` ▶ Head The first element of (the no empty) sequence `s`.
Xmetoo	`(hd s)`
Example	`hd(<"ann","peter","bill">)` $\equiv$ `"ann"`
Semantics	hd s

`tl(_)`

Syntax Rule	$\langle LIST\rangle$ ::= `tl(`$\langle LIST\rangle$`)`
Description	`tl(s)` ▶ Tail The sequence obtained from (the no empty) sequence `s` by removing its first element.
Xmetoo	`(tl s)`
Example	`<hd(<"a","p","b">) : tl(<"a","p","b">)>` $\equiv$ `<"a","p","b">`
Semantics	tl s

`nth(_,_)` [_[_]]

Syntax Rule	$\langle S\rangle$ ::= `nth(`$\langle INT\rangle$, $\langle LIST\rangle$`)`
Description	`nth(i,s)` ▶ Element by position The element in position `i` of (the no empty) sequence `s`.
Xmetoo	`(nth i s)`
Example	`nth(2,<"ann","peter","bill">)` $\equiv$ `"peter"`
Semantics	$s[i]$, a sequence being defined as a function from an initial segment of $\overline{n}$ of $\mathbb{N}$ to an arbitrary set

---

`plus(_, _)`

Syntax Rule	$\langle LIST\rangle$ ::= $\langle LIST\rangle$ + $\langle FF\rangle$
Description	`plus(s,m)` ▶ Overwrite The sequence obtained from `s` by overwriting elements in each position `i` belonging to the domain of `m`, by `m[i]`, iff `i` belongs to the domain of the original sequence `s`.
Xmetoo	`(plus s m)`
Example	`plus(<"ann","sue","bill">, [2->"bill",1->"sue"])` $\equiv$ `<"sue","bill","bill">`
Semantics	$s \dagger m$

`length(_)`

Syntax Rule	$\langle \mathit{INT} \rangle$ ::= `length(`$\langle \mathit{LIST} \rangle$`)`
Description	`length(s)` ▶ Length The number of elements of sequence `s`.
Xmetoo	`(length s)`
Example	`length(<2,3,5,2,3,4,1,1,2,6>)` $\equiv$ `10`
Semantics	$\#\,\mathrm{dom}\ s$

`elems(_)`

Syntax Rule	$\langle \mathit{SET} \rangle$ ::= `elems(`$\langle \mathit{LIST} \rangle$`)`
Description	`elems(s)` ▶ Set of elements The set formed by the elements of sequence `s`.
Xmetoo	`(elems s)`
Example	`elems(<2,3,5,2,3,4,1,1,2,6>)` $\equiv$ {`2,3,5,4,1,6`}
Semantics	$\{x \mid x \leftarrow s\}$

`inds(_)`

Syntax Rule	$\langle \mathit{LIST} \rangle$ ::= `inds(`$\langle \mathit{LIST} \rangle$`)`
Description	`inds(s)` ▶ Indices The sequence of natural numbers used as domain of sequence `s`.
Xmetoo	`(inds s)`
Example	`inds(<2,3,5,2,3,4,1,1,2,6>)` $\equiv$ `<1,2,3,4,5,6,7,8,9,10>`
Semantics	$\overline{\#\,\mathrm{dom}\ s}$ seen as a sequence

---

`_ -seq`

Syntax Rule	$\langle \mathit{FUN} \rangle$ ::= $\langle \mathit{FUN} \rangle$`-seq`
Description	`f-seq` ▶ Functorial extension of a function The function that applies `f` to each element of the (list) argument.
Xmetoo	-
Example	`succ-seq(<1,2,3>)` $\equiv$ `<2,3,4>`
Semantics	f^*

`_-red(_,_)` [`reduce(_,_,_)`]

SYNTAX RULE	⟨*S*⟩ ::= ⟨*SYM*⟩`-red(` ⟨*S*⟩ `,` ⟨*LIST*⟩`)`
DESCRIPTION	`o-red(e,s)` ▶ Distributed form Applies to sequence `s` the distributed form of the associative binary function `o` whose unit element is `e`.
XMETOO	`(reduce e o s)`
EXAMPLE	`.+-red(0,<1,2,3,3,2,1>)` ≡ `12`
SEMANTICS	See similar operation on finite sets.

2.5 Finite Binary Relations

`[**]`

SYNTAX RULE	$\langle REL \rangle$ ::= {}
DESCRIPTION	`[**]`
	▶ Empty relation
	Constant standing for the empty relation.
XMETOO	`nil`
EXAMPLE	`[* x -> x .* 7	x <- {5,2} : x >= 10 *]` ≡ `[**]`
SEMANTICS	$\emptyset$, as binary relations are defined as sets of pairs.

`_U_` [`union(_,_)`]

SYNTAX RULE	$\langle REL \rangle$ ::= $\langle REL \rangle$ `U` $\langle REL \rangle$
DESCRIPTION	`s U r`
	▶ Binary relation union
	The relation formed by union of relations `s` and `r`.
XMETOO	`(union s)`
EXAMPLE	`[* "a"->12, "p"->13 *] U [* "p"->15 *]` ≡ `[* "a"->12, "p"->13, "p"->15 *]`
SEMANTICS	$s \cup r$

`_*_` [`intersection(_,_)`]

SYNTAX RULE	$\langle REL \rangle$ ::= $\langle REL \rangle$ `*` $\langle REL \rangle$
DESCRIPTION	`s * r`
	▶ Binary relation intersection
	The relation formed by intersection of relations `s` and `r`.
XMETOO	`(intersection s r)`
EXAMPLE	`[* "a"->5, "a"->12, "p"->13 *] * [* "p"->15, "b"->9 *]` ≡ `[**]`
SEMANTICS	$s \cap r$

`_-_` [`difference(_,_)`]

SYNTAX RULE	$\langle REL \rangle$ ::= $\langle REL \rangle$ `-` $\langle REL \rangle$
DESCRIPTION	`s - r`
	▶ Binary relation difference
	The relation formed from `s` by removing all the pairs common to relation `r`.
XMETOO	`(difference s r)`
EXAMPLE	`[* "a"->5, "a"->12, "p"->13 *] - [* "p"->13 *]` ≡ `[* "a"->5, "a"->12 *]`
SEMANTICS	$s - r$

`[*_->_,_->_...,_->_*]`

Syntax Rule	$\langle REL \rangle$ ::= `[*` $\langle S \rangle$ `->` $\langle S \rangle$ [(`,` $\langle S \rangle$ `->` $\langle S \rangle$)+] `*]`
Description	`[d1->x1, ..., dn->xn)]` ▶ Binary relation enumeration The relation consisting of pairs `d1->x1`, `d2->x2`, ..., `dn->xn`.
Xmetoo	`(makerel (d1 x1) (d2 x2) ... (dn xn))`
Example	`[* "ann"->12, "peter"->8, "ann"->14 *]`
Semantics	$\{(d_1, x_1), ..., (d_n, x_n)\}$

`[*_ -> _|_<-_,...:_*]`

Syntax Rule	$\langle REL \rangle$ ::= `[*` $\langle S \rangle$ `->` $\langle S \rangle$ `|` $\langle SYM \rangle$ `<-` $\langle SET \rangle$ [(`,` $\langle SYM \rangle$ `<-` $\langle SET \rangle$)+] [`:` $\langle BOOL \rangle$] `*]`
Description	`[* ed -> er | x <- s : p *]` ▶ Binary relation comprehension The binary relation resulting from applying expressions `ed` and `er` to those elements of set `s` which verify predicate `p`. The expressions `ed`, `er` and `p` are, in general, dependent on the variable `x`.
Xmetoo	`(rel (pair ed er) (from x s p) ...)`
Example	`[* x -> abs(x) | x <- {1,-2,2,-4} *]` $\equiv$ `[* 1->1,-2->2,2->2,-4->4 *]`
Semantics	$\{ed(x_1, ..., x_n) - > er(x_1, ..., x_n) \| \; x_1 \in s_1, ..., x_n \in s_n \; \wedge p(x)\}$

`_ == [**]` [`isempty(_)`]

Syntax Rule	$\langle BOOL \rangle$ ::= $\langle REL \rangle$ `== [**]`
Description	`s == [**]` ▶ Empty relation test True if `s` is the empty relation.
Xmetoo	`(isempty s)`
Example	`[* 1 -> {"a","b"}, 2 -> {} *] == [**]` $\equiv$ `false`
Semantics	$s = \emptyset$

`subset(_,_)`

Syntax Rule	$\langle BOOL \rangle$ ::= `subset(`$\langle REL \rangle$`,` $\langle REL \rangle$`)`
Description	`subset(r,s)` ▶ Relation inclusion True if `r` is a subset of relation `s`.
Xmetoo	`(subset r s)`
Example	`subset([*"ann" -> 5 *],[* "ann" -> 5, "ann" -> 12 *])` $\equiv$ `true`
Semantics	$r \subseteq s$

_ [card(_)]

SYNTAX RULE	⟨*INT*⟩ ::= `#` ⟨*REL*⟩
DESCRIPTION	`# s` ▶ Cardinal The cardinal number of relation `s`
XMETOO	`(card s)`
EXAMPLE	`# [* "ann" -> 5, "ann" -> 12 *]` ≡ `2`
SEMANTICS	$\#s$

dom(_)

SYNTAX RULE	⟨*SET*⟩ ::= `dom(`⟨*REL*⟩`)`
DESCRIPTION	`dom(r)` ▶ Domain The domain of the relation `r`.
XMETOO	`(dom r)`
EXAMPLE	`dom([* "ann" -> 5, "ann" -> 12, "peter" -> 11 *])` ≡ `{"ann","peter"}`
SEMANTICS	dom r

ran(_)

SYNTAX RULE	⟨*SET*⟩ ::= `ran(`⟨*REL*⟩`)`
DESCRIPTION	`ran(r)` ▶ Co-Domain The co-domain of the relation `r`.
XMETOO	`(ran r)`
EXAMPLE	`ran([* "ann" -> 5, "ann" -> 12, "peter" -> 11 *])` ≡ `{5,12,11}`
SEMANTICS	rng r

_ @ _ [ds(_,_)]

SYNTAX RULE	⟨*REL*⟩ ::= ⟨*REL*⟩ `@` ⟨*SET*⟩
DESCRIPTION	`r @ s` ▶ Domain subtraction The relation obtained from `r` by removing the set `s` from the domain of `r`.
XMETOO	`(ds r s)`
EXAMPLE	`[* "ann"->5, "ann"->12, "sue"->11 *] @ {"ann"}` ≡ `[* "sue"->8 *]`
SEMANTICS	$\{(x, r(x)) \mid x \in (\text{dom } m - s)\}$

_ / _ [dr(_,_)]

SYNTAX RULE	⟨*REL*⟩ ::= ⟨*REL*⟩ `/` ⟨*SET*⟩
DESCRIPTION	`r / s` ▶ Domain restriction The relation obtained from `r` by restricting the domain of `r` to the set `s`.
XMETOO	`(dr r s)`
EXAMPLE	`[* "ann"->5, "ann"->12, "sue"->1 *] / {"ann"}` ≡ `[* "ann"->5, "ann"->12 *]`
SEMANTICS	$\{(x, r(x)) \mid x \in (\mathrm{dom}\ r \cap s)\}$

2.6 Integers & Strings

`_ .+ _` [`add(_,_)`]

SYNTAX RULE	$\langle INT \rangle$ ::= $\langle INT \rangle$ `.+` $\langle INT \rangle$
DESCRIPTION	`n .+ m` ▶ Addition The result of adding integers `n` and `m`.
XMETOO	`(add n m)`
EXAMPLE	`52 .+ 5` $\equiv$ `57`
SEMANTICS	$n + m$

`_ .- _` [`sub(_,_)`]

SYNTAX RULE	$\langle INT \rangle$ ::= $\langle INT \rangle$ `.-` $\langle INT \rangle$
DESCRIPTION	`n .- m` ▶ Subtraction The result of subtracting integer `m` from `n`.
XMETOO	`(sub n m)`
EXAMPLE	`52 .- 5` $\equiv$ `48`
SEMANTICS	$n - m$

`_ .* _` [`mul(_,_)`]

SYNTAX RULE	$\langle INT \rangle$ ::= $\langle INT \rangle$ `.*` $\langle INT \rangle$
DESCRIPTION	`n .* m` ▶ Multiplication The result of multiplying integers `n` and `m`.
XMETOO	`(mul n m)`
EXAMPLE	`52 .* 5` $\equiv$ `260`
SEMANTICS	$n \times m$

`_ ./ _` [`div(_,_)`]

SYNTAX RULE	$\langle INT \rangle$ ::= $\langle INT \rangle$ `./` $\langle INT \rangle$
DESCRIPTION	`n ./ m` ▶ Integer division The result of dividing integers `n` and `m`.
XMETOO	`(div n m)`
EXAMPLE	`52 ./ 5` $\equiv$ `10`
SEMANTICS	$n \div m$

`rem(_,_)`

SYNTAX RULE	⟨*INT*⟩ ::= `rem(` ⟨*INT*⟩ `,` ⟨*INT*⟩ `)`
DESCRIPTION	`rem(n,m)` ▶ Reminder of integer division The reminder of dividing integers `n` and `m`.
XMETOO	`(rem n m)`
EXAMPLE	`rem(52,5)` ≡ `2`
SEMANTICS	$\mathrm{mod}(n, m)$

`abs(_)`

SYNTAX RULE	⟨*INT*⟩ ::= `abs(` ⟨*INT*⟩ `)`
DESCRIPTION	`abs(n)` ▶ Absolute value The absolute value of integer `n`.
XMETOO	`(abs n)`
EXAMPLE	`abs(-52)` ≡ `52`
SEMANTICS	$\mathrm{abs}(n)$

`ascii(_)`

SYNTAX RULE	⟨*INT*⟩ ::= `ascii(` ⟨*STR*⟩ `)`
DESCRIPTION	`ascii(s)` ▶ Converter The ASCII representation of (the unit) string `s`.
XMETOO	`(ascii s)`
EXAMPLE	`ascii("a")` ≡ `97`

`chr(_)`

SYNTAX RULE	⟨*STR*⟩ ::= `chr(` ⟨*INT*⟩ `)`
DESCRIPTION	`chr(n)` ▶ Converter The string with ASCII code `n`.
XMETOO	`(chr n)`
EXAMPLE	`chr(97)` ≡ `"a"`

`atoi(_)`

Syntax Rule	$\langle INT \rangle$ ::= `atoi(` $\langle STR \rangle$ `)`
Description	`atoi(s)` ▶ Converter The string `s`, or its numeric prefix, seen as an integer; returns `0` in error condition.
Xmetoo	`(atoi s)`
Example	`atoi("934asd9")` ≡ `943`

`itoa(_)`

Syntax Rule	$\langle STR \rangle$ ::= `itoa(` $\langle INT \rangle$ `)`
Description	`itoa(n)` ▶ Converter The integer `n` seen as a string.
Xmetoo	`(itoa n)`
Example	`atoi(itoa(91242))` ≡ `91242`

`symstr(_)`

Syntax Rule	$\langle STR \rangle$ ::= `symstr(` $\langle SYM \rangle$ `)`
Description	`symstr(s)` ▶ Converter The symbol `s` seen as a string.
Xmetoo	`(symstr s)`
Example	`symstr('o)` ≡ `"o"`

`strsym(_)`

Syntax Rule	$\langle SYM \rangle$ ::= `strsym(` $\langle STR \rangle$ `)`
Description	`strsym(s)` ▶ Converter The string `s` seen as a symbol.
Xmetoo	`(strsym s)`
Example	`strsym("o")` ≡ `'o`

`strcat(_,_,...,_)` [_ ++ _]

SYNTAX RULE	⟨*STR*⟩ ::= `strcat(` ⟨*STR*⟩ [(`,` ⟨*STR*⟩)+] `)`
DESCRIPTION	`strcat(s1,s2,...,sn)` ▶ String concatenation The string formed by concatenation of strings `s1` to `sn`.
XMETOO	`(strcat s1 s2 ... sn)`
EXAMPLE	`strcat("ann ","mary ","harisson")` $\equiv$ `"ann mary harisson"`
SEMANTICS	$s1 \frown s2 \frown ... \frown sn$

`substr(_,_,_)`

SYNTAX RULE	⟨*STR*⟩ ::= `substr(` ⟨*STR*⟩ `,` ⟨*INT*⟩ `,` ⟨*INT*⟩ `)`
DESCRIPTION	`substr(s,i,j)` ▶ Sub-string The sub-string of string `s` from position `i` to `j`.
XMETOO	`(substr s i j)`
EXAMPLE	`substr("ann mary harisson",4,10)` $\equiv$ `" mary hari"`

`strlen(_)`

SYNTAX RULE	⟨*INT*⟩ ::= `strlen(` ⟨*STR*⟩ `)`
DESCRIPTION	`strlen(s)` ▶ String length The length of the string `s`.
XMETOO	`(strlen s)`
EXAMPLE	`strlen("ann mary harisson")` $\equiv$ `17`

`_ <_` [`lt(_,_)`]

SYNTAX RULE	⟨*BOOL*⟩ ::= (⟨*STR*⟩ | ⟨*INT*⟩) `<` (⟨*STR*⟩ | ⟨*INT*⟩)
DESCRIPTION	`x < y` ▶ Less than The result of comparing `x` and `y`.
XMETOO	`(lt x y)`
EXAMPLE	`"ann" < "peter"`
SEMANTICS	$x < y$

`_ <= _` [`le(_,_)`]

Syntax Rule	$\langle BOOL\rangle$ ::= ($\langle STR\rangle$ | $\langle INT\rangle$) `<=` ($\langle STR\rangle$ | $\langle INT\rangle$)
Description	`x <= y` ▶ Less than or equal to The result of comparing `x` and `y`.
Xmetoo	`(le x y)`
Example	`"ann" <= "ann"`
Semantics	$x \leq y$

`_ > _` [`gt(_,_)`]

Syntax Rule	$\langle BOOL\rangle$ ::= ($\langle STR\rangle$ | $\langle INT\rangle$) `>` ($\langle STR\rangle$ | $\langle INT\rangle$)
Description	`x > y` ▶ Greater than The result of comparing `x` and `y`.
Xmetoo	`(gt x y)`
Example	`9 > -9`
Semantics	$x > y$

`_ >= _` [`ge(_,_)`]

Syntax Rule	$\langle BOOL\rangle$ ::= ($\langle STR\rangle$ | $\langle INT\rangle$) `>=` ($\langle STR\rangle$ | $\langle INT\rangle$)
Description	`x >= y` ▶ Greater than or equal to The result of comparing `x` and `y`.
Xmetoo	`(ge x y)`
Example	`555 >= 90`
Semantics	$x \geq y$

`rand(_)`

Syntax Rule	$\langle INT\rangle$ ::= `rand(`$\langle INT\rangle$`)`
Description	`rand(n)` ▶ Random generator A number between `0` and `n-1`, randomly generated.
Xmetoo	`(rand n)`
Semantics	$rand(n) \in \overline{n-1}$

2.7 Logical Connectives

_ /@_ [_&&_] [and(_,_)]

SYNTAX RULE	$\langle BOOL \rangle$ ::= $\langle BOOL \rangle$ `/@` $\langle BOOL \rangle$
DESCRIPTION	`p /@ q` ▶ Conjunction
XMETOO	`(and p q)`
SEMANTICS	$s \wedge r$

_@/ _ [_||_] [or(_,_)]

SYNTAX RULE	$\langle BOOL \rangle$ ::= $\langle BOOL \rangle$ `@/` $\langle BOOL \rangle$
DESCRIPTION	`p @/ q` ▶ Disjunction
XMETOO	`(or p q)`
SEMANTICS	$s \vee r$

~ _ [not(_)]

SYNTAX RULE	$\langle BOOL \rangle$ ::= `~` $\langle BOOL \rangle$
DESCRIPTION	`~ p` ▶ Negation
XMETOO	`(not p)`
SEMANTICS	$\neg p$

_ => _

SYNTAX RULE	$\langle BOOL \rangle$ ::= $\langle BOOL \rangle$ `=>` $\langle BOOL \rangle$
DESCRIPTION	`p => q` ▶ Implication The formal implication of `q` by `p`.
XMETOO	`(or (not p) q)`
SEMANTICS	$s \Rightarrow r$

`all(_ <- _, ... : _ )`

SYNTAX RULE	$\langle BOOL \rangle$ ::= `all(` $\langle SYM \rangle$ `<-` $\langle SET \rangle$ [(, $\langle SYM \rangle$ `<-` $\langle SET \rangle$)+] `:` $\langle BOOL \rangle$
DESCRIPTION	`all(x <- s : p)` ▶ Universal quantifier Predicate `p` is verified by every element of set `s`.
XMETOO	`(ALL s e ... p)`
EXAMPLE	`all(x <- {2,4,6,8}, y <- {1,3,5} : rem(x .+ y,2) != 0)` $\equiv$ `true`
SEMANTICS	$\forall_{x_1 \in s_1, \ldots, x_n \in s_n} \cdot p(x_1, ..., x_n)$

`exist(_ <- _ , ... : _ )`

SYNTAX RULE	$\langle BOOL \rangle$ ::= `exist(` $\langle SYM \rangle$ `<-` $\langle SET \rangle$ [(, $\langle SYM \rangle$ `<-` $\langle SET \rangle$)+] `:` $\langle BOOL \rangle$
DESCRIPTION	`exist(x <- s : p)` ▶ Existential quantifier Predicate `p` is verified by at least one element of set `s`.
XMETOO	`(EXIST s e ... p)`
EXAMPLE	`exist(x <- {2,4,6,8}, y <- {1,4,5} : rem(x .+ y,2) != 0)` $\equiv$ `true`
SEMANTICS	$\exists_{x_1 \in s_1, \ldots, x_n \in s_n} \cdot p(x_1, ..., x_n)$

`exist1(_ <- _ , ... : _ )`

SYNTAX RULE	$\langle BOOL \rangle$ ::= `exist1(` $\langle SYM \rangle$ `<-` $\langle SET \rangle$ [(, $\langle SYM \rangle$ `<-` $\langle SET \rangle$)+] `:` $\langle BOOL \rangle$
DESCRIPTION	`exist1(x <- s : p)` ▶ Exclusive existential quantifier Predicate `p` is verified by one and only one element of set `s`.
XMETOO	`(EXIST1 s e ... p)`
EXAMPLE	`exist1(x <- {2,4,6,8}, y <- {1,4,5} : rem(x .+ y,2) != 0)` $\equiv$ `false`
SEMANTICS	$\exists^1_{x_1 \in s_1, \ldots, x_n \in s_n} \cdot p(x_1, ..., x_n)$

2.8 Structural Comparators

`_ == _` [`eq(_,_)`]

SYNTAX RULE	$\langle BOOL \rangle$ ::= $\langle S \rangle$ `==` $\langle S \rangle$
DESCRIPTION	`x == y` ▶ Equality The result of testing for structural equality the expressions `x` and `y`.
XMETOO	`(eq x y)`
EXAMPLE	`{x	x <- {1,2,3} : x>1} == {2,3}` ≡ `true`
SEMANTICS	$x = y$

`_ != _` [`neq(_,_)`]

SYNTAX RULE	$\langle BOOL \rangle$ ::= $\langle S \rangle$ `!=` $\langle S \rangle$
DESCRIPTION	`x != y` ▶ Inequality The result of testing for structural inequality the expressions `x` and `y`.
XMETOO	`(neq x y)`
EXAMPLE	`{1,2,3} != <1,2,3>` ≡ `true`
SEMANTICS	$x \neq y$

2.9 Expressions

---

if _ then _ else _

SYNTAX RULE	$\langle S\rangle$ ::= if $\langle BOOL\rangle$ then $\langle S\rangle$ [else $\langle S\rangle$]
DESCRIPTION	if b then e1 else e2
	▶ Evaluates condition b returning the result of evaluating expression e1 or e2 as b evaluates to true or false, respectively.
EXAMPLE	if (x<=10) then x else x .* 10

---

if (_ -> _ , ...) otherwise(_)

SYNTAX RULE	$\langle S\rangle$ ::= if($\langle BOOL\rangle$ -> $\langle S\rangle$ [(, $\langle BOOL\rangle$ -> $\langle S\rangle$)+]) [otherwise ($\langle S\rangle$)]
DESCRIPTION	if (b1 -> e1, ..., bn -> en) otherwise(s)
	▶ Evaluates expression ei if condition bi is verified. If all the guards (b1 to bn) fail returns the result of expression s. If otherwise is ommited and all the guards fail returns NIL
EXAMPLE	if (x<=10 -> x, (x<=100) /@ (x>10) -> x .+ 10) otherwise("invalid x")

---

if (_ -> _ , ..., else -> _)

SYNTAX RULE	$\langle S\rangle$ ::= if($\langle BOOL\rangle$ -> $\langle S\rangle$ [(, $\langle BOOL\rangle$ -> $\langle S\rangle$)+] [, else -> $\langle S\rangle$])
DESCRIPTION	if (b1 -> e1, ..., bn -> en, else -> s)
	▶ Evaluates expression ei if condition bi is verified. If all the guards (b1 to bn) fail returns the result of expression s. NIL is returned as above.
EXAMPLE	if (x<=10 -> x, (x<=100) /@ (x>10) -> x .+ 10, else -> "invalid x")

NOTE:

A restricted form of structural condition, which additionaly provides a local binding, is available as guards in conditional expressions, as shown in the table below.

Condition	Semantics
x is-<>	Verifies if x is a empty sequence
x is-<h:t>	Verifies if x is a no empty sequence and creates a local binding of h to hd(x) and t to tl(x)
x is-{}	Verifies if x is a empty set
x is-{o:d}	Verifies if x is a no empty set and creates a local binding of o to choice(x) and d to x - {o}
x is-[]	Verifies if x is a empty mapping
x is-[k->i:d]	Verifies if x is a no empty mapping and creates a local binding of k to choice(dom(x)), i to x[k] and d to x @ {o}

`let (_ = _ , ...) in _`

SYNTAX RULE ⟨*S*⟩ ::= `let(` ⟨*SYM*⟩ `=` ⟨*S*⟩ [(`,` ⟨*SYM*⟩ `=` ⟨*S*⟩)+] `in` ⟨*S*⟩]

DESCRIPTION `let (s1 = e1, ..., sn = en) in e`

▶ Evaluates expression `e` in a context enriched with the local definitions of symbols `s1` to `sn`. A minor syntactical modification allows for the structural decomposition of a pair of values as in, for example, `let (<a,b> = p) in ...`. If `p` is defined as a pair, the identifiers `a` and `b` will become instatiated with its first and second component, respectively.

EXAMPLE `let (f = lambda(x), x .+ 1, s = {5,6,7}) in {f(x) | x<-s }`

`_(_ ...)`

SYNTAX RULE ⟨*S*⟩ ::= ⟨*FUN*⟩ | ⟨*SUBR*⟩ `(` [⟨*S*⟩] [(`,` ⟨*S*⟩)+] `)`

DESCRIPTION `f(a1, ..., an)`

▶ Applies function `f` to arguments `a1` to `an`.

EXAMPLE `dom([])` ≡ `{}`

`eval(_)`

SYNTAX RULE ⟨*S*⟩ ::= `eval(` ⟨*S*⟩ `)`

DESCRIPTION `eval(s)`

▶ Forces evaluation of expression `e`.

EXAMPLE `eval(<add,2,2>)` ≡ `4`

`quote(_)` `´_`

SYNTAX RULE ⟨*S*⟩ ::= `quote(` ⟨*S*⟩ `)`

DESCRIPTION `quote(s)`

▶ Prevents expression `e` from being evaluated. Notice that the alternative syntax may be used without parentthesis only when `´` is aplied to values of type `SYM`. Therefore valid uses are, for example, `´name` and `´(2 .+ 9)`.

EXAMPLE `eval(quote(dom([]))` ≡ `dom([])`

`_ _o_ _`

Syntax Rule	⟨*FUN*⟩ ::= ⟨*FUN*⟩ `_o_` ⟨*FUN*⟩
Description	`f _o_ g` ▶ Functional composition (`f` after `g`).
Example	`(reverse _o_ reverse) (<1,2,3>)` ≡ `<1,2,3>`

`_c _`

Syntax Rule	⟨*FUN*⟩ ::= `_c` ⟨*ANY*⟩
Description	`_c v` ▶ Returns the constant function, i.e., the function that outputs the value `v` everywhere.
Example	`_c "error" (25)` ≡ `"error"`

`id-_`

Syntax Rule	⟨*FF*⟩ ::= `id-` ⟨*SET*⟩
Description	`id-A` ▶ Given a finite set `A`, returns the identity function on this set, represented as a finite function.
Example	`id-{1,2,3,4,5} [3]` ≡ `3`

`sh(_)`

Syntax Rule $\langle INT \rangle$ ::= `sh(` [$\langle STR \rangle$] `)`

Description `sh(s)`

▶ Executes command included in the string `s` under the operating system shell, returning the command error code. In case `s` is omitted an operating system shell is created on top of CAMILA.

Example `sh("ls -la > ftemp")`

`foreach(_,_,_)`

Syntax Rule $\langle S \rangle$ ::= `foreach(` $\langle SYM \rangle$ `,` ($\langle LIST \rangle$ | $\langle SET \rangle$ | $\langle REL \rangle$ | $\langle FF \rangle$) `,` $\langle S \rangle$ [(`,` $\langle S \rangle$)+] `)`

Description `foreach(x,l,e1,...,en)`

▶ Expressions `e1` to `en` are evaluated for each element `x` in sequence `l`. The value of the last evaluated expression is returned.

Example `foreach(x,<1,2,3,4>,x.+1)` ≡ `5`

`while(_,_)`

Syntax Rule $\langle S \rangle$ ::= `while(` $\langle BOOL \rangle$ [`,` $\langle S \rangle$ [(`,` $\langle S \rangle$)+]] `)`

Description `while(b,l)`

▶ Expressions `e1` to `en` in sequence `l` are evaluated while `b` is `true`. The last evaluated expression or `NIL` (if `b` is `false` in the first time or `l` omitted) is returned.

Example `while(read() != 0,procMenu(),chooseOption)`

`type(_)`

Syntax Rule $\langle SYM \rangle$ ::= `type(` $\langle S \rangle$ `)`

Description `type(e)`

▶ Returns the type of expressions `e`.

Example `type([8->"a",6->"b"])` ≡ `FF`

`atom(_)`

Syntax Rule $\langle BOOL \rangle$ ::= `atom(` $\langle S \rangle$ `)`

Description `atom(e)`

▶ Returns `true` if expressions `e` is of type `INT`, `STR`, `SYM`, `BOOL`, `FPTR`, `CHAN` or `SUBR`.

Example `atom([8->"a",6->"b"])` ≡ `false`

2.10 I/O Primitives

`fclose(_)`

Syntax Rule	$\langle \mathit{NIL} \rangle$ ::= `fclose(` $\langle \mathit{FPTR} \rangle$ `)`
Description	`fclose(p)`
	▶ Returns `NIL`; as a side effect closes the file pointed by `p`.

`getc(_)`

Syntax Rule	$\langle \mathit{INT} \rangle$ | $\langle \mathit{NIL} \rangle$ ::= `getc(` $\langle \mathit{FPTR} \rangle$ `)`
Description	`getc(p)`
	▶ Returns the Ascii code of the next character in the file pointed by `p` or `NIL` if end-of-file is detected.

`fgets(_)`

Syntax Rule	$\langle \mathit{SRT} \rangle$ | $\langle \mathit{NIL} \rangle$ ::= `fgets(` $\langle \mathit{FPTR} \rangle$ `)`
Description	`fgetc(p)`
	▶ Returns the next line in the file pointed by `p` or `NIL` if end-of-file is detected.

`fopen(_ , _)`

Syntax Rule	$\langle \mathit{FPTR} \rangle$ | $\langle \mathit{NIL} \rangle$ ::= `fopen(` $\langle \mathit{STR} \rangle$, $\langle \mathit{STR} \rangle$ `)`
Description	`fopen(s,m)`
	▶ Returns a file pointer to the file identified by `s` which is open in mode `m`. The following modes are available: `"r"` (read only), `"w"` (overwrite), `"a"` (append), `"r+"` (read and write in an already created file), `"w+"` (read or write in a new file) and `"a+"` (read or write after the end of file). Returns `NIL` if an error occurs.

---

putc(_ , _)

Syntax Rule	⟨*INT*⟩ ::= `putc(` ⟨*INT*⟩ [`,` ⟨*FPTR*⟩] `)`
Description	`putc(n,p)`
	▶ Returns `n`; as a side effect writes `n` in the file pointed by `p` or, if `p` is omitted, on the current output channel.

fputs(_ , _)

Syntax Rule	⟨*STR*⟩ ::= `fputc(` ⟨*STR*⟩ [`,` ⟨*FPTR*⟩] `)`
Description	`fputs(s,p)`
	▶ Returns `s`; as a side effect writes `s` in the file pointed by `p` or, if `p` is omitted, on the current output channel.

m2read(_)

Syntax Rule	⟨*S*⟩ ::= `m2read(` ⟨*FPTR*⟩ `)`
Description	`m2read(p)`
	▶ Returns the evaluation of the S-expression read in the file pointed by `p`.

m2write(_ , _)

Syntax Rule	⟨*NIL*⟩ ::= `m2write(` ⟨*FPTR*⟩ `,` ⟨*S*⟩ `)`
Description	`m2write(p,e)`
	▶ Returns `NIL`; as a side effect writes the evaluation of expression `e` in the file pointed by `p`.

nwrite(_ , _)

Syntax Rule	⟨*NIL*⟩ ::= `nwrite(` ⟨*FPTR*⟩ `,` ⟨*S*⟩ `)`
Description	`nwrite(p,e)`
	▶ Returns `NIL`; as a side effect writes in the Camila notation the evaluation of expression `e` in the file pointed by `p`.

`stdin`

SYNTAX RULE	⟨*FPTR*⟩ ::= `stdin`
DESCRIPTION	`stdin`
	▶ Returns a pointer to the current standard input.

`stdout`

SYNTAX RULE	⟨*FPTR*⟩ ::= `stdout`
DESCRIPTION	`stdout`
	▶ Returns a pointer to the current standard output.

`popen(_ , _)`

SYNTAX RULE	⟨*FPTR*⟩ | ⟨*NIL*⟩ ::= `popen(` ⟨*STR*⟩ `,` ⟨*STR*⟩ `)`
DESCRIPTION	`popen(s,m)`
	▶ Returns a pointer to a UNIX pipe created with the command `s` which is open in mode `m` (modes are described in the entry `fopen`).

`pclose(_)`

SYNTAX RULE	⟨*NIL*⟩ ::= `pclose(` ⟨*FPTR*⟩ `)`
DESCRIPTION	`pclose(p)`
	▶ Returns `NIL`; as a side effect closes the UNIX pipe pointed by `p`.

`load(_)`

SYNTAX RULE	⟨*STR*⟩ ::= `load(` ⟨*STR*⟩ `)`
DESCRIPTION	`load(s)`
	▶ Returns `s`; as a side effect reads and evaluates `xmetoo` definitions on file named `s` .

`LOAD(_)`

SYNTAX RULE	⟨*STR*⟩ ::= `LOAD(` ⟨*STR*⟩ `)`
DESCRIPTION	`LOAD(s)`
	▶ Returns a message; as a side effect builds the internal state of the prototyping environment from an object file named `s` .

`nload(_)`

SYNTAX RULE	⟨*STR*⟩ ::= `nload(` ⟨*STR*⟩ `)`
DESCRIPTION	`nload(s)`
	▶ Returns a message; as a side effect reads and evaluates CAMILA definitions on file named `s` .

`print(_ ,_ ...)`

SYNTAX RULE	⟨*NIL*⟩ ::= `print(` ⟨*S*⟩ [(, ⟨*S*⟩)+] `)`
DESCRIPTION	`print(e1,...,en)`
	▶ Returns `NIL`; as a side effect displays the expressions `e1` to `en`.

`princ(_ ,_ ...)`

SYNTAX RULE	$\langle NIL \rangle$ ::= `princ(` $\langle S \rangle$ [(, $\langle S \rangle$)+] `)`
DESCRIPTION	`princ(e1,...,en)`
	▶ Returns `NIL`; as a side effect displays the expressions `e1` to `en`; the usual control commands for newline, tab, etc, are correctly handled.

`read(_)`

SYNTAX RULE	$\langle S \rangle$ ::= `read(` [$\langle STR \rangle$] `)`
DESCRIPTION	`read(s)`
	▶ Returns the expression extracted from string `s`; if `s` is omitted reads an expression from the keyboard.

`stload(_)`

SYNTAX RULE	$\langle S \rangle$ ::= `stload(` $\langle STR \rangle$ `)`
DESCRIPTION	`stload(s)`
	▶ Returns the expression stored in the file named `s`. Typical use: restoring the value of a state variable from a previous testing session.

`stsave(_)`

SYNTAX RULE	$\langle NIL \rangle$ ::= `stsave(` $\langle STR \rangle$, $\langle S \rangle$ `)`
DESCRIPTION	`stsave(s,e)`
	▶ Returns `NIL`; as a side effect the value of expression `e` is saved in a file named `s`. Typical use: saving the values of a state variable after a testing session.

`SAVE(_)`

SYNTAX RULE	$\langle NIL \rangle$ ::= `SAVE(` $\langle STR \rangle$ `)`
DESCRIPTION	`SAVE(s)`
	▶ Returns `NIL`; as a side effect the internal state of the prototyping environment is saved in an object file named `s`.

2.11 Communication Primitives

`channel(_,_)`

SYNTAX RULE $(\langle CHAN\rangle \mid \langle NIL\rangle)$ ::= `channel(` $\langle STR\rangle$ [`,` $\langle INT\rangle$] `)`

DESCRIPTION `channel(s,n)`

▶ Returns a communication channel named `s` or `NIL` if an error occurs. The optional parameter `n` is the session identifier — the contents of `XMSESSION` by default.

`receive(_)`

SYNTAX RULE $\langle S\rangle$ ::= `receive(` $\langle CHAN\rangle$ `)`

DESCRIPTION `receive(c)`

▶ Returns the expression received on channel `c`.

`rmchannel(_)`

SYNTAX RULE $\langle NIL\rangle$ ::= `rmchannel(` $\langle CHAN\rangle$ `)`

DESCRIPTION `rmchannel(c)`

▶ Returns `NIL`; as a side effect closes channel `c`.

`send(_ ,_)`

SYNTAX RULE $\langle NIL\rangle$ ::= `send(` $\langle CHAN\rangle$ `,` $\langle S\rangle$ `)`

DESCRIPTION `send(c,e)`

▶ Returns the result of evaluating expression `e`; as a side effect sends this value on channel `c`.

---

3 The Prototyping Environment

3.1 Using Camila

The Camila prototyping tool is basically a *calculator* of Camila expressions running on a *read-eval-print* cycle, similar to what can be found in other functional programming interpreters. Its development was guided by two main objectives

- *Simplicity.* As a tool for software engineers and programmers, a small, easy to work with kernel, with a sober interface, was favored rather than a heavy and sophisticated interface layer which would compromise its *availability.* Once constructive set theory has been chosen as a specification language and a simple (Ascii-) notation devised for it, the prototyper was designed as a processor for that notation, offering additionally just that collection of support primitives considered indispensable in practice
- *Connectivity.* Some (technical) sophistication on the prototyping environment was introduced to ensure its full connectivity with standard programming languages and systems, bridging the gap between the specification level and implementation development. The reader is referred to [AB94] or [BA99] for a full account of this topic, which is not documented here.

This being stated, the key topics in using the prototyping tool are briefly described in what follows.

From the operating system level, the prototyping environment is called by the command `camila`. The prompt (`?-`) appears and, from that point on, the tool is ready to evaluate any Camila expression or to accept any definitions of global variables (here called objects) or functions. Accepting a definition amounts to create an association, for further reference, between an identifier and an expression. For example,

```
S <- {"ann","peter","mary"};
succ <- lambda(x). x .+ 1;
```

The contents of such bindings is easily displayed by typing the respective identifier followed, as usual, by a semi-colon. For example `S;` or `succ;`. The list of current associations can be inspected at any time looking the contents of the global variables `obs` (for object definitions) or `ops` (for user defined functions). The function `types`, with no arguments, does the same for type definitions.

Usually the sets of types, functions and state variables definitions of a particular specification are maintained in one or more text files which are loaded for animation and test. This can be done when calling Camila, as in

```
camila enroll1.cam auxdef.cam
```

or at any time during the prototyping session, using the `nload` primitive, as in,

```
?-  nload("enroll1.cam","auxdef.cam");
```

In both cases the Camila definitions in the files are checked for parsing errors and, in its absence, added to the environment current state. Trying an alternative version of a definition during the session is accomplished by the primitives `vi`, to edit, save and load a file (as in `vi "enroll1.cam"`, or `edit`, to edit the contents of an object (as in `edit(S)`). This is particularly useful for directly forcing a specific configuration in a state variable.

The state variables themselves can be saved and restored independently using the primitives `stsave` and `stload`, a feature which turns out to be relevant whenever a particular configuration of a component state becomes interesting to be tested on alternative versions of a design or in different contexts. For example, assuming `S` as the current state variable, the following functions may be defined to be called when a testing session is over and at the beginning of a new session, respectively,

```
FUNC done()
RETURNS stsave("Prototype1.State",S);

FUNC restore()
STATE  S <- stload("Prototype1.State");
```

Finally, the internal state of a CAMILA prototyping session, which includes all the definitions made, can be saved as an object file (using `SAVE`) and later restored (using `LOAD`). The following table sums up the most useful support functions available.

SUPPORT PRIMITIVES	
`obs`	Lists the current object definitions
`ops`	Lists the current function definitions
`sig()`	Shows the signatures of all functions defined with `FUNC`
`types()`	Lists the current type definitions
`nload(l)`	Compiles and loads a list of CAMILA files
`load(s)`	Compiles and loads a xmetoo file
`stsave(f,e)`	Saves the value of expression `e` in file `f`
`stload(f)`	Restores the value of a definition stored in file `f`
`SAVE(f)`	Saves the current environment state in object file `f`
`LOAD(f)`	Restores the current environment state from object file `f`
`vi(f)`	Calls VI to edit file `f` and load it
`cat(filename)`	Displays the contents of file `f`
`edit(o)`	Edits (with VI) the values of object `o`
`!()`	Returns to the Operating System

Some further features of the prototyping environment are,

- **Global objects**. Besides the state variables, which are inherent to the semantics of a software component, other auxiliary variables, with a global scope, can be defined at any time for handling intermediate results or structured values for testing. They are defined, at any level, by `X <- ... ;`.
- **Undefined functions**. Whenever CAMILA is asked to evaluate a function for which a definition has not been supplied the interpreter will ask the user for a value. With this feature a prototype can be tested, once defined its *interface*, from the very initial stages of a design.

- **Costumazing**. The prototyping environment can be adapted to meet the specific needs of a particular project by placing in the configuration file `.metoorc` definitions of functions or objects to be loaded whenever the `camila` command is executed. Such definitions can range from the trivial change of the default text editor (which is VI) to the less trivial inclusion of test and analyze facilities.For example, the enhanced IO primitives described below, which do not belong to the tool kernel, are supplied in the standard configuration file. To modify the current setting of `.metoorc`, edit the file `camilarc`, compile it into `xmetoo` format (`seca camilarc > .metoorc`) and copy the new `.metoorc` file to your home directory.
- **Including**. Any text file containing CAMILA definitions may include other similar files, at any point, using the command `#include` followed by the included file identifier.

3.2 The Command Line Interface

The CAMILA prototyping environment incorporates the GNU Readline Library which provides a powerful command line interface. In this section some of its features are commented (please refer to the Library Manual for further details).

Often during an interactive session you type in a long line of text, only to notice that the first word on the line is misspelled. The Readline library provides a set of commands for manipulating the text as it is typed in, allowing to fix typos, and not forcing to retype the majority of the line. Using these editing commands, you move the cursor to the place that needs correction, and delete or insert the text of the corrections. Then, when you are satisfied with the line, you simply press `RET`. You do not have to be at the end of the line to press `RET`; the entire line is accepted regardless of the location of the cursor within the line.

A list of the basic bare essentials for editing the text of an input line follows. Notice C-b means the key CTRL followed by the character b and M-x stands for the metafont key followed by x (the same keystroke can be generated by typing ESC first and then x).

C-b Move back one character.

C-f Move forward one character.

DEL Delete the character to the left of the cursor.

C-d Delete the character underneath the cursor.

Printing characters Insert itself into the line at the cursor.

C-_ Undo the last thing that you did. You can undo all the way back to an empty line.

C-a Move to the start of the line.

C-e Move to the end of the line.

M-f Move forward a word.

M-b Move backward a word.

C-l Clear the screen, reprinting the current line at the top.

"Killing" text means to delete the text from the line, but to save it away for later use, usually by "yanking" it back into the line. When use is made of a kill command, the text is saved in a "kill-ring". Any number of consecutive kills save all of the killed text together, so that when it is yanked back, you get it in one clean sweep. The kill ring is not line specific; the text killed on a previously typed line is available to be yanked back later, when typing another line. Here is the list of commands for killing and yanking text.

C-k Kill the text from the current cursor position to the end of the line.

M-d Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

M-DEL Kill from the cursor the start of the previous word, or if between words, to the start of the previous word.

C-w Kill from the cursor to the previous whitespace. This is different than M-DEL because the word boundaries differ.

C-y Yank the most recently killed text back into the buffer at the cursor.

M-y Rotate the kill-ring, and yank the new top. You can only do this if the prior command is C-y or M-y.

And, finally, the following are some commands for manipulating the editing history

RET Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list. If this line was a history line, then restore the history line to its original state.

C-p Move 'up' through the history list.

C-n Move 'down' through the history list.

C-r Search backward starting at the current line and moving 'up' through the history as necessary. This is an incremental search.

C-s Search forward starting at the current line and moving 'down' through the the history as necessary.

3.3 Enhanced I/O

Besides the CAMILA primitives for Input/Output from and to files (namely the standard input and output devices) and pipes described in the Reference Manual (Section 2), some additional functions are declared in the configuration file — `.metoorc` — and, therefore, automatically loaded when the prototyping tool is called. They consist of a set of facilities for pretty-printing expressions, structuring and handling menus and reading with a user defined prompt.

These features may help in improving the interface of a prototype in order to create a more friendly test environment. However, in any case shall this be confused with the process of prototyping the user interface of a particular system. The purpose of the CAMILA prototyping environment is to prototype the *semantics* of a system design and not its user interface (which should be specified and, eventually, prototyped, independently). The next table sums up these additional primitives, whose use is illustrated with a few examples.

ENHANCED I/O	
`pp(e)`	Pretty-print of expression `e`
`lerstr(p)`	Interactive read of a string with a string prompt `p`
`lerint(p)`	Interactive read of an integer with a string prompt `p`
`lerset(p)`	Interactive read of a set with a string prompt `p`
`lersetstr(p)`	Interactive read of a set of strings with a string prompt `p`
`lerdixdomff(s)`	Given a set `s`, seen as the domain of a mapping, interactive read of the images, returning the mapping
`pickone(x)`	Interactive choice of an element from a set, a sequence, a binary relation or a finite function `x`
`pickmany(x)`	Interactive choice of part of a set, a sequence, a binary relation or a finite function `x`
`ppickone(x,p)`	Same as `pickone(x)` but using a string prompt
`menu(m)`	Display of menu `m` and selection of an option
`run(m:tmenu)`	Execution of menu `m` until selection of an option equal to 0

Examples

The first example is a function `interIns` which allows the interactive reading of student's identifiers and marks in a state based version of the *Enrollment System* prototype discussed in [BA99]. The state variable is assumed to be `C` of type `Course`.

```
...
interIns() = InsMark(lerstr("Student"), lerint("Mark"));
...
```

```
?- interIns();
 Student  > "peter"
 Mark  > 14
C
?-
```

Next consider a menu interface to run all the operations available in the prototype. A variable `M` of `tmenu` type is instantiated with a title and a list of menu options. Every option has a title and an action to be performed. Each action is quoted to be evaluated only after selection. The type `tmenu` is a product collecting all the information relevant to a menu, which means that a product constructor with the same identifier is available.

```
...
M <- tmenu( "Enrollment System",
           <opcoes("Insert a student mark", '(interIns()) ),
            opcoes("Remove a student ", '(RemStu(lerstr("name? "))) ),
            opcoes("Show course ", '(C)),
            opcoes("Did better than ... ", '(DidBetter(lerint("than: ")))),
```

```
          );
   ...
```

A sample session is shown below.

```
?- run(M)
--------------------------------------------------
    Enrollment System

    1 Insert a student mark
    2 Remove a student
    3 Show course
    4 Did better than ...

option?  3

--------------------------------------------------
 [ "ann" -> 12
   "peter" -> 14
   "george" -> NIL
   "john" -> 9 ]
 {}

 --------------------------------------------------
    Enrollment System

    1 Insert a student mark
    2 Remove a student
    3 Show course
    4 Did better than ...

 option?  1

 --------------------------------------------------
  Student > "george"
  Mark  > 13

  C

 --------------------------------------------------
    Enrollment System

    1 Insert a student mark
    2 Remove a student
    3 Show course
    4 Did better then ...

 option?  0

 ?-
```

3.4 Camila & xmetoo

The Camila prototyping tool runs on top of a functional environment — called xmetoo [Pin89, The90]— developed earlier during the project. Therefore every Camila expression is codified in xmetoo as an

S-expression with an assigned type. Although in normal use this low level machine is hidden from the user, it can be useful to resort to it to increase performance or link to external applications.

xmetoo may be called directly from the operating system level, by the command `xmetoo`. Additionally a list of text files with xmetoo source code may be specified, in which case a "load" statement is executed for each of them. On the other hand xmetoo code can be generated from a CAMILA prototype using the `seca` translator.

The xmetoo interpreter manages an internal stack storing the expressions whose evaluation is not yet completed. If the evaluation fails, the stack is shown on the screen, starting with the expression in which the error occurred and including all the previous calling levels. The user may control the step-by-step evaluation of any S-expression by the activation of an internal flag (called the trace-flag) using the primitive function "trace".

A CAMILA shell is also available in which all the interaction with the prototypes is done in the xmetoo notation (it is the realm of the parenthesis ...). The shell is called by the command `vcamila` followed by the file name of the prototype file to be loaded. Its specification is mandatory.

References

[AB94] J. J. Almeida and L. S. Barbosa. From Prototypes to Implementations and Back. Technical Report DI/INESC-94-10-2, INESC (U. Minho), 1994.

[ABBN98] J. J. Almeida, L. S. Barbosa, J. B. Barros, and L. F. Neves. On the development of CAMILA. In L. S. Barbosa and J. A. Saraiva, editors, *Workshop on Research Themes on Functional Programming*. UM, 18 Sep. 1998.

[ABNO97a] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. CAMILA: Formal software engineering supported by functional programming. In A. De Giusti, J. Diaz, and P. Pesado, editors, *Proc. II Conf. Latino Americana de Programación Funcional (CLaPF97)*, pages 1343–1358, La Plata, Argentina, October 1997.

[ABNO97b] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. CAMILA: Prototyping and refinement of constructive specifications. In M. Johnson, editor, *6th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, pages 554–559, Sydney, December 1997. Springer Lect. Notes Comp. Sci. (1349).

[BA99] L. S. Barbosa and J. J. Almeida. Systems prototyping in CAMILA. Lecture Notes for the Bristol Course (1st ed. 1995) DI-CAM-95:11:1:v99, DI (U. Minho), 1999.

[Hen84] P. Henderson. me too: A language for software specification and model building. Preliminary Report, University of Stirling, 1984.

[Pin89] A. Pinto. Desenvolvimento de xmetoo: Um Núcleo para Prototipagem Rápida de Especificações VDM. Relatório de Estágio da LESI, Universidade do Minho, 1989.

[The90] The CAMILA Group. xmetoo User's Manual. Technical Report, INESC (U. Minho), 1990.