Reasonable Lisp

Research Report LIX.RR.90.01, pp 37–84, École Polytechnique, January 1990, France.

Christian Queinnec*
Internet: queinnec@poly.polytechnique.fr
Laboratoire d'Informatique de l'École Polytechnique
91128 Palaiseau Cedex — France

Working DRAFT as of September 14, 1989 at 11:53
Sendai Issue

This document is a trial to define a reasonable Lisp language. Its design spirit was to precisely describe a slightly extended kernel of currently used Lisps in order to provide answers to semantic questions such as evaluation order, evaluation time or evaluation environment. The main aspect of this work was to select an harmonious and efficient bunch among all interesting features that have been experimented for many years in many Lisp implementations. Many design decisions might have been far more ambitious, but we thought that it was preferable to live in a "deterministic" Lisp, whatever it may be, rather than in a poorly semantically defined Lisp with many trouble spots. On the other hand we tried to give a reasonable meaning to many fuzzy but useful constructs such as types, modules, macros, exceptions and threads.

Care have also been taken to only design the "run-time" part of Lisp i.e. to clearly separate environmental features such as `trace`, `*evalhook*` or `remove-method` ...from execution primitives and to exclude the former features: programs must not use environmental features to be run. We put a great emphasis not to be encumbered by costly run-time data; symbols are, for instance, nearly eliminated from run-time. We also try not to reject interpretation nor the possibility of development of production environments.

The form of the document is also an attempt of mixing formal semantics with description text. All essential linguistic features are described by a commented denotation. Other parts such as library functions or useful macros are not present, since they do not introduce semantical problems: they surely must be defined but the means to define them (Lisp itself) are already present in this document. This work is left to implementors or to users.

Reasonable Lisp will be recognized by many to be in the mainstream of MacLisp offspring. Despite the mathematical substract, that have often been stressed by Lisp, a precisely defined semantics was still waited for. Scheme has one [Rees & Clinger 86], parts of Lisp also has some [Gordon 75, Muchnick & Pleban 80] but all these semantics do not describe complex features such as `load` or `eval-when`. Our attempt here is to provide the semantics of a richer Lisp with these questionable and maybe unavoidable features. We put great efforts in producing in this document a precise semantics for features which are fuzzy by their very nature: modules, macros, exceptions, threads are described.

We have to acknowledge many influences both linguistic from Ada[1], Common Lisp, Le-Lisp[2], LTR3, Modula-3, Scheme, T ...and from many people ...

*This work has been partially funded by Greco de Programmation and Ministère de la Recherche et de l'Enseignement Supérieur (contrat MRES 88 S 1094).

[1] Ada is a trademark of the Ada Joint Program Office.

[2] Le-Lisp is a trademark of INRIA.

Contents

1 Terminology

The semantic descriptions of this document stick (or try to) to a common terminology as developped below. The first use of such terms is usually italicized: that means that an entry defining this term appears here.

environment

Environments record *bindings* between various things: *names*, *values*, *continuations*, *locations* ... Environments often preexist with a null initial binding collection.

binding

Bindings map *keys* (usually names) to *informations* (usually locations). Bindings are created via *defining forms* (e.g. `defmodule`) or *binding forms* (e.g. `catch`). These forms confer them a *scope*, an *extent* and a *mutability*.

Given a key, a binding may be accessed and deliver the corresponding information. A binding may be captured (closed) in order to be reused elsewhere.

A binding may be referenced: only its existence is important, not its value though that will probably be read or written afterthat. A binding closure is a binding reference. A binding may also be read and yield the associated information or written and vary the associated information.

context

A context gathers a number of environments.

The *lexical context* is composed of all lexical environments i.e. the lexical variable environment, the lexical multiple argument environment, the lexical functional environment, the lexical escape environment, the type environment and the scheduler environment.

Similarly, the *dynamic context* is composed of the dynamic environment and the dynamic escape environment.

The current context gathers the lexical and the dynamic context.

scope

The scope of a binding is the textual region where the binding may be referenced thanks to a key. Once referenced a binding may be read or written depending on the kind of operations that are legal on the binding. Two scopes exist in Reasonable Lisp. A binding has a *dynamic scope* if it can be referenced from anywhere. A binding has a *lexical scope* if introduced by a binding form and limited to some texts enclosed by this binding form. Lexical bindings exported from modules allow their lexical scope to be extended to the body of the modules which import them.

extent

The extent of a binding is its lifetime. Two extents exist. A binding has a *dynamic extent* if created when entering a binding form and destroyed after exiting it. The binding may only be used during the computation of the binding form and cannot be closed. A binding with *indefinite extent* is created by a binding form and will disappear only if useless. It is the duty of the Garbarge Collector to collect useless bindings. The extent is indefinite not infinite ! Closure analyses may determine a more precise behaviour for such bindings.

mutability

Some bindings are mutable. Given a binding with a key, the information part may be varied. The binding is not altered per se but only mutated. Other bindings are immutable, constant folding is thus possible i.e. a reference with a known key may be replaced by the information. An immutable binding does not preclude subparts of the information to be altered.

object

Objects are first class values that are both computable and storable. Numbers, structures, functions are objects. They all have a type.

entity

Entities stand for things that are off-the-scenes and not directly accessible. They are usually handled by syntax or special forms. Continuations, bindings ... are entities.

functional or parametric

Two places are distinguished in a form: the functional place, in the `car` of the form and the parametric place, elsewhere in the `cdr` of the form. According to the position, the handling of a name differs. A property of Reasonable Lisp is that only names can appear in a functional place. The expressions appearing in parametric place are called parameters. When the form is not a special form i.e. has a function in functional place, then the values of its parameters are called its arguments and are bound to the variable(s) of the function.

evaluators

Reasonable Lisp is a Lisp_2 [Gabriel & Pitman 88]. In fact Reasonable Lisp would be more precisely called a Lisp_{11} since there are other environments where names can appear with different semantics.

defining form

Forms may be defining forms or not. Defining forms may only appear in the toplevel of modules (they may also appear in such a position after macroexpansion). Except for mutable global lexical variables, all defining forms defines immutable bindings that cannot be mutated (nor redefined): types, modules, functions... are examples of such immutable bindings.

error

Two kinds of errors may appear. When an erroneous situation is recognized and an exception is raised, the user has a chance to regain control. Such situations are described in this document and provide the necessary information: when the exception is raised, wether it is continuable and its very class. Conversely the words "is an error" (typically a domain error for a function) mean that the situation is erroneous but the implementation is not required to detect this situation. The implementation is therefore free to react with the most erratic manner (and even to crash). Many of these situations may be easily avoided since some predicates are provided to detect them. Conversely, every potentially erroneous situation which may be avoided if adequately tested before will be labelled as an error in this document. Exceptions are reserved for unpredictable situations only.

Many situations which are described in this document would probably never appear because they may syntactically be always recognized: for instance, an undefined global lexical reference may always be recognized thanks to the module semantics. Pre-evaluators such as compilers are free to emit warnings for them. But if they output something, that result must behave as indicated in this document i.e. raise an exception or react erratically.

others

compound object, visible, name, value, continuation, location, key, scope, extent, mutability ...

special form

Special forms may be implemented as macros and reversely. Some precautions are to be caught...

2 Overall Features

Following Lisp tradition, there are numerous environments. Numerous environments allow strict separation of groups of features. For instance, lexical variables are clearly separated from dynamic escapes[3]. Related special forms may thus be described independently. Furthermore, to have such different spaces allow the evaluation process to fully determine how some hidden entities are precisely handled and thus allow some

[3] That property will be used in denotational equations, see section 3

kind of optimizations. Not confering these entities a first-class citizenship relieves evaluators from complex analyses to determine their intended behaviour which may be diluted in a variety of places and not strictly confined by a lexical fence. On the other hand numerous environments burden the programmer with plethoric evaluation schemata that all these spaces provide.

If Lisp systems are kind of λ-calculus, the set of special forms they offer makes the main difference between the various dialects. REASONABLE LISP has many specialized special forms. Although not complex they often exist to ensure a small run-time size or to be an efficient way to handle off-the-scenes entities. Special forms are to be distinguished from defining forms which, as their name suggests it, defines some entities or objects by extension of global environments. Defining forms are static in the sense that they cannot be computed (but can be obtained by macroexpansion). If the precise definition of the entity which will be bound to a name is usually only obtained after a computation which will take place when modules are loaded, the characteristics of the new binding are known at module definition time. Defining forms appear to be keywords introducing definitions inside modules while special forms are forms which must follow precise rule for evaluation. Defining forms are not evaluated even if some part of it is, they do not yield a value. Special forms and defining forms of REASONABLE LISP are

progn	if	quote	
fixed-arity-lambda	setq		
multiple-arity-lambda	mv-call	mv-length	mv-ref
mv-set	mv-values		
flet	labels		
dynamic-bind	dynamic-ref	dynamic-set	
	loadmodule		
block	return-from	catch	throw
		unwind-protect	
type			with-handler
schedule	suspend	resume	
	Special Forms		

constant-define	variable-define	defvar	
defun	defgeneric	defmethod	
defmodule	export		
deftype	defmaker	defgetter	defsetter
	Defining Forms		

In order to give a flavor of REASONABLE LISP, all environments are shortly described hereafter.

Lexical Variable

This environment binds names to locations.

initial environment: null.
binding creation: `lambda` application (and macros such as `let`).
binding reading: *name* in parametric position.
binding writing: `setq`
binding mutability: mutable if `setq` appears in the scope, immutable if not.
binding scope: lexical
binding extent: indefinite
context: lexical

Multiple Argument

This environment binds names to sequences of locations.

initial environment:	null.
binding creation:	`multiple-arity-lambda` application.
binding reading:	`(mv-ref` *name index*`)`.
binding writing:	`(mv-set` *name index value*`)`
binding mutability:	mutable if `mv-set` appears in the scope, immutable if not.
binding scope:	lexical
binding extent:	indefinite
context:	lexical

Global Lexical Variable

This environment binds names to locations.

initial environment:	null. There exists one global lexical variable environment per module.
binding creation:	`constant-define`, `variable-define`. Global bindings cannot be re-created.
binding reading:	*name* in parametric position. It is forbidden to refer to a global lexical variable which has not been defined.
binding writing:	`setq` if the binding is mutable.
binding mutability:	mutable if created by `variable-define`, immutable if created by `constant-define`.
binding scope:	the whole module (and other modules via exportation).
binding extent:	indefinite
context:	lexical

Functional

This environment binds names to functions.

initial environment:	null.
binding creation:	`flet`, `labels`
binding reading:	*name* in functional position or as parameter of `function`.
binding writing:	none
binding mutability:	always immutable
binding scope:	lexical
binding extent:	indefinite
context:	lexical

Global Functional

This environment binds names to functions.

initial environment:	null. There exist one global functional environment per module.
binding creation:	`defun`, `defgeneric`
binding reading:	*name* in functional reference or as parameter of `function`.
binding writing:	none
binding mutability:	always immutable
binding scope:	the whole module (or other modules via exportations).
binding extent:	indefinite
context:	lexical

Dynamic

This environment binds objects to locations. No connexion with symbols.

initial environment:	null.
binding creation:	`dynamic-bind`
binding reading:	`dynamic-ref`
binding writing:	`dynamic-set`
binding mutability:	always mutable.
binding scope:	indefinite
binding extent:	dynamic
context:	dynamic

Lexical Escape

This environment binds names to continuations.

initial environment:	null.
binding creation:	`block`
binding reading:	`return-from`
binding writing:	none
binding mutability:	immutable
binding scope:	lexical
binding extent:	indefinite (but the continuation may only be used without exception in the dynamic extent of `block`).
context:	lexical

Dynamic Escape

This environment binds values to continuations.

initial environment:	null.
binding creation:	`catch`
binding reading:	`throw`
binding writing:	none
binding mutability:	none
binding scope:	indefinite
binding extent:	dynamic
context:	dynamic

Type

This environment binds names to types.

initial environment:	Some predefined types such as `cons`, `symbol` ...There exists one type environment per module.
binding creation:	`defclass`
binding reading:	as first parameter of `type`
binding writing:	none
binding mutability:	none
binding scope:	module (or others via exportation)
binding extent:	indefinite
context:	lexical

Scheduler

This environment binds names to schedulers.

initial environment:	null
binding creation:	`schedule`
binding reading:	as first parameter of `suspend`
binding writing:	none
binding mutability:	none
binding scope:	lexical
binding extent:	dynamic
context:	lexical

Module

This environment binds names to modules.

initial environment:	the standard modules.
binding creation:	`defmodule`
binding reading:	`loadmodule` and importation clauses.
binding writing:	none.
binding mutability:	always immutable.
binding scope:	indefinite
binding extent:	indefinite
context:	not contextual

3 Denotational Conventions

Denotations that appear in this document follow one of two different forms. The first one is classical and makes use and abuse of small greek letters. It is precise but obscures the main lines by far too much details: denotational λ-terms with dozens of variables are not easy to read nor to write. Moreover many of these variables are given to continuations as they are and without any changes. Another notation is provided which makes use of lexically scoped abbreviations. Abbreviated equations only show important parts which are either read, changed or extended. For example in many Lisp dialects, the meaning of an alternative is defined as

```
(defmeaning (alternative condition then else)
  (build :meaning M
     (call :meaning (meaning condition)
           :cont (build :cont C
                    (if (convert-to-boolean C.value)
                        (call :meaning (meaning then))
                        (call :meaning (meaning else)) ) ) ) ) )
```

First the *condition* is evaluated and its result is given to the continuation which, depending on its logical value, evaluates the *then* or the *else* part of the original alternative. More denotationally, we will have in Scheme something which looks like

$$
\begin{array}{l}
meaning-of-ALTERNATIVE= \\
\lambda\ \pi_1\pi_2\pi_3\ . \\
\quad \lambda\ \rho_m\gamma_m\sigma_m\kappa_m\ . \\
\quad\quad (\mathcal{E}[\![\pi_1]\!])(\rho_m,\gamma_m,\sigma_m,\ \lambda\ \varepsilon_c\gamma_c\sigma_c\ . \\
\quad\quad\quad\quad\quad\quad \mathbf{if}\ convert-to-boolean(\varepsilon_c) \\
\quad\quad\quad\quad\quad\quad \mathbf{then}\ (\mathcal{E}[\![\pi_2]\!])(\rho_m,\gamma_c,\sigma_c,\kappa_m) \\
\quad\quad\quad\quad\quad\quad \mathbf{else}\ (\mathcal{E}[\![\pi_3]\!])(\rho_m,\gamma_c,\sigma_c,\kappa_m) \\
\quad\quad\quad\quad\quad\quad \mathbf{endif}\ )
\end{array}
$$

The above `defmeaning` form is first expanded into

```
(defmeaning (alternative condition then else)
  (meaning-lambda (M.lexenv M.globalenv M.store M.cont)
     ((meaning condition)
      (the-current-globalenv)
      (the-current-store)
      (cont-lambda (C.value C.globalenv C.store)
          (if (convert-to-boolean C.value)
            ((meaning then) (the-current-lexenv)
                            (the-current-globalenv)
                            (the-current-store)
                            (the-current-cont) )
            ((meaning else) (the-current-lexenv)
```

```
                                  (the-current-globalenv)
                                  (the-current-store)
                                  (the-current-cont) ) ) ) ) ) )
```

All missing parameters are inserted where needed. `build` forms are converted into λ-terms whilst `call` forms are turned into appropriate combinations. All the current denotational entities referred as `(the-current-`*something*`)` are looked up in the lexical context. `(the-current-cont)` then refers to the closest visible cont: `M.cont`. The final complete denotation is then

```
(defmeaning (alternative condition then else)
  (lambda (M.lexenv M.globalenv M.store M.cont)
     ((meaning condition)
      M.lexenv M.globalenv M.store
      (lambda (C.value C.globalenv C.store)
          (if (convert-to-boolean C.value)
            ((meaning then) M.lexenv C.globalenv C.store M.cont)
            ((meaning else) M.lexenv C.globalenv C.store M.cont) ) ) ) ) )
```

These abbreviations are quite comfortable and closely correspond to denotational habits. The reader (or designer) of such abbreviated equations is directly presented the important things. Similarly fully denotational equations can be displayed with important variables underlined to ease their identification (see other equations in this document for examples).

Another advantage of this approach is that it is now very simple to modify the signature of domains since the expansion takes care of the arguments order and nature. Equations are then simple to reuse, for example, the previous abbreviated equation giving the meaning of an alternative could have been the same[4] in REASONABLE LISP but would have been expanded into

$$
\begin{array}{l}
meaning.of.ALTERNATIVE = \\
\lambda\ \pi_1\pi_2\pi_3\ . \\
\quad \lambda\ \rho_m\gamma_m\delta_m\eta_m\zeta_m\sigma_m\kappa_m\ . \\
\quad\quad (\mathcal{E}[\![\pi_1]\!])(\rho_m, \gamma_m, \delta_m, \eta_m, \zeta_m, \sigma_m,\ \lambda\ \underline{\varepsilon_c^*}\gamma_c\underline{\eta_c}\sigma_c\ . \\
\quad\quad\quad \textbf{if}\ convert-to-boolean(\underline{\varepsilon_c^*} \downarrow 1) \\
\quad\quad\quad \textbf{then}\ (\mathcal{E}[\![\pi_3]\!])(\rho_m, \gamma_c, \delta_m, \underline{\eta_c}, \zeta_m, \sigma_c, \kappa_m) \\
\quad\quad\quad \textbf{else}\ (\mathcal{E}[\![\pi_2]\!])(\rho_m, \gamma_c, \delta_m, \eta_c, \zeta_m, \sigma_c, \kappa_m) \\
\quad\quad\quad \textbf{endif}\ )
\end{array}
$$

One can see here that superfluous (and orthogonal) domains are just passed through or ignored when needed.

3.1 Denotational Notations

The conventions for denotational equations are taken from [Rees & Clinger 86] except that conditional (`if`) and local binding (`let`) will be written in a more readable way. The conventions are

$\sharp e$	`(LENGTH E)`
$<>$	`(LIST)`
$< a, b, c >$	`(LIST A B C)`
$a \downarrow i$	`(ELT A I)`
$a \S b$	`(APPEND A B)`
$a \dagger 1$	`(REST A)`
$a[b \rightarrow c]$	`(EXTEND A B C)`
$a \wedge b$	`(AND A B)`
$a \vee b$	`(OR A B)`

3.2 Abbreviated Equation Conventions

The conventions for abbreviated equations are taken from Lisp. `defmeaning` defines semantics for syntactic constructs...

[4] Except for multiple values which induces changes while passing a value to a continuation.

A denotational λ-term is built by

(`build` :*domain label body*)

where *domain* is the name of a functional domain (such as `meaning`, `cont` or `lexenv` ...), *label* is a label which allows to name the variables of the correspondingly built λ-term. The generic names of the variables are taken from the tags which appear in the domain definition (see below). The precise names of the variables of the built λ-term are obtained by the concatenation of the label, a dot and the generic name. The *body* is the body of the λ-term and may also use the whole set of lexical abbreviations.

A denotational λ-term may be applied by

(`call` :*domain* λ*-term [keyworded-argument...]*)

where *domain* is the name of a functional domain, λ*-term* is the denotational function to apply and must, of course, belong to *domain*. *keyworded-arguments* are pairs made of a keyword and an argument associated to the parameter indicated by the keyword. Parameters can contain abbreviations. Names of keywords are taken from the tags that appear in *domain* definition. Since keyworded, parameters do not need to be ordered (remember that λ-calculus is side-effect free).

An abnormal situation is represented as a call to `wrong`

(`wrong` *[keyworded-argument...]*)

The precise expansion depends on the definition of `wrong`

A denotational object is built by

(`make-`*domain [keyworded-argument...]*)

Keywords are taken from the names of the tag fields of the domain referred as *domain*. *keyworded-arguments* may appear in any order.

The expansion of all these expressions heavily depends on the domain definitions and on the names which are used in them. Domains are defined by `def-domain`

(`def-domain` *name domain-expression*)

where *domain-expression* must respect the following grammar

```
domain-expression
       ::= ( = Natural ) ;a domain isomorphic to the set of naturals
        |  ( = Symbol )  ;a domain isomorphic to the set of symbols
        |  ( = String )  ;a domain isomorphic to the set of symbols
        |  ( * domain-expression ... )    ;;Cartesian Product
        |  ( -> domain-expression domain-expression ... ) ;;Function
        |  ( + domain-expression ... )    ;;Disjoint union
        |  ( kleene domain-expression )      ;;list of
        |  ( label name domain-expression ) ;;local name
```

4 Syntactical Conventions

Description of the entries.

The names of the parameters are choosen to illustrate the values they represent. Table 4 contains the used names and their meanings.

form	any object	*thread*	a thread object
variable	an identifier	*exception*	an exception object
function	a functional object	*type-name*	a name of a type
function-name	a name of a function	*type-descriptor*	a type descriptor
natural	a natural number	*slot-name*	a name of a slot
number	a number		
index	a natural number	*module-name*	a name of a module

5 Program and Syntax

Programs are written using a very simple syntax. The syntax only requires a few characters: left and right parentheses, quote, backquote, comma and dot and some other characters. Space also separates tokens. This representation looks like S-expressions and may be handled by macros. Therefore and as usual, forms are represented by lists and lexical variables by symbols. Other objects such as numbers, strings or keywords stand for themselves.

Since keywords appear in program representations and since that representation may be computed upon by macros: keywords are first class objects. Keywords form a type irrelated to symbols. Keywords have only a name. Every fixed arity function is "keywordisable" i.e. can be called with the keywords which bear the names of its variables. Predefined functions therefore expose the names of their variables in order to be called with keywords. In a keyworded call, keywords are always related to the function which appears in functional position (see `funcall`).

6 Types

All computed values have a type. Some predefined types exist for convenience among which are integers, characters and references (ako pointers). Types and classes are distinct concepts but are both created by `deftype`. Classes are a special kind of types which are linked to the class hierarchy. Types are not related to the class hierarchy except that some types may be extended into classes: `Object`, the root of all objects, `Dotted-Pair` ... are extensible whilst `Function` is not.

The type system is powerful, it allow to describe a wide range of objects in a uniform manner while retaining efficiency on usual aggregates such as structures, vectors, matrices ...

Generic functions allow to gather specific behaviours which will be selected according to the types of their arguments. New generic functions can be created by `defgeneric` and new methods can be added to these generic functions thanks to `defmethod`. Both forms are defining forms and may only appear in the toplevel of modules.

The resulting class system is very simple, offers simple inheritance and simple discrimination: features that are all well understood and very efficient. Types are handled by their name and form the type environment. Type descriptors can be obtained through `type-of`. Types may be coerced into type descriptors thanks to the special form `type`. Type descriptors are the run-time support of types; the only operations that can be applied on them are comparison and subtype relation. When types are defined, a set of special forms allow to define readers, writers and allocators for these types.

Types may be exported. The module which imports a type, knows its structure and can define appropriate readers, writers and allocator for it. A module can also export a limited set of functions encapsulating the type which is then opaque.

6.1 Type Related Domains

Something like

```
(def-domain "Value" (+ Boolean
                       Empty
                       Proc
                       Pair
                       (label num (= Integer))
                       (label data Id)    ; Quote operate only on symbols
                       Thread
                       TypeDesc
                       Object
                       ))
(def-domain "TypeDesc" (is altlocation))
(def-domain "Object" (* Type Locations))
(def-domain "Type" (* TypeDesc TypeDef))
```

```
(def-domain "TypeDef" (+ (label Basic (enum integer reference))
                         (Times (Kleene Typedefs))
                         (Power (* Natural Typedef))
                         (Star Typedef)
                         (Named (* Id Typedef))
                         (Extended (* Typedef Typedefs)) ) )
(def-domain "TypeDefs" (kleene typedef))
```

6.2 Type General Features

Function **type-of**

(**type-of** *form*)

Every computable object has a unique type. For every object, the function `type-of` returns the appropriate *type descriptor*, a kind of run-time projection of types that may be compared with equality or subtype relation.

Predicate **type-descriptor-p**

(**type-descriptor-p** *form*)

Like every object, type descriptors have types but, in some implementations, they may have different types. It is only ensured that all type descriptors return true if submitted to `type-descriptor-p`.

(type-descriptor-p (type-of (cons 1 2))) → *true*
(type-descriptor-p (type-of (type-of (cons 1 2)))) → *true*

Predicate **type-eq**

(**type-eq** *form*$_1$ *form*$_2$)

Type descriptors may be compared for equality with this specialized predicate. The result is a boolean. It is an error to compare with `type-eq` objects which are not type descriptors (as checked by `type-descriptor-p`).

```
(type-eq (type-of (type-of (cons 1 2)))
         (type-of (type-of 3.14)) )
```
; *may return true or false depending on the implementation.*

Special Form **type**

(**type** *type-name*)

This special form returns the type descriptor associated with the type named *type-name*. There is only one type descriptor associated to a given type i.e. two invokations of `type` on *type-name* return the same type-descriptor (while in the same program execution, type descriptors are not persistent nor any other object of REASONABLE LISP). Visible predefined types such as `thread`, `exception` ... are also supported by type descriptors and may be coerced into them.

It is now simple to provide appropriate type predicate for every type. For example

```
(defun thread-p (exp)
  (type-eq (type-of exp) (type thread)) )
```

Defining Form **deftype**

(**deftype** *name type-description*)

This special form defines a new type named *name* in the current type environment. The binding is immutable and is visible everywhere in the module. The type may be exported and made visible in other modules.

The structure of the type is defined by the *type-description* which must obey the following grammar

type-description ::= bit | character | fixnum | float | any ;; *Basic types*

```
| type-name                 ;;other visible types
| (times type-description ...)
| (power form type-description)
| (star type-description)
| (extend type-name type-description ...)
| (label name type-description)
```

Basic types are the usual ones and are those generally and natively offered by computers. Data may be aggregated with appropriate type composers

`times`	concatenates the representations (like records)
`power`	concatenates a fixed number of representations (like fixed vectors)
`star`	repeats a variable number of representations (like strings)
`extend`	extend a class with additional slots
`label`	gives a label to a component of a type-description, useful to easily refer to subparts of a type.

Although *type-description* looks like an expression, it is only a description of the structure of the type and therefore is not evaluated.

Here are some examples of types

```
(deftype Point (times (label x integer) (label y integer)))
(deftype Point3D (power 3 integer))
(deftype String (star character))
(deftype PackedList (star reference))
(deftype Matrix (power (dynamic-ref '*size*) (power 10 integer)))
;;A class definition for Point
(deftype Point (extend Object (times (label x integer) (label y integer))))
(deftype ColoredPoint (extend Point (label color reference)))
```

Classes can be defined by extension of previously defined classes. A predefined class exist which name is `Object`. The `Object` class has no fields (no slots) and is the root of the inheritance as used by generic functions. Another predefined class is `Exception`. Among predefined types are `thread`, `symbol`, `dotted-pair`, `fixnum`, `boolean` ... A class is a type but not all types are classes. Classes are related with the subclass relation, types are not related to the class hierarchy. Types which are not classes cannot be extended.

Defining Form **defgetter**

(`defgetter` *name type-name* {*natural* |*slot-name* |_} ...)

This form defines a reader to a precise subpart of instances of *type-name*. The subpart is described by a path formed by the third and following parameters of the `defgetter` form. The path may be given with numbers, names (if the corresponding subpart was given a label in the `deftype` form) or may also be the special symbol _. In the lattest case the missing index will be given numerically at invokation time. The reader is a function named *name*. That function takes as many arguments as there are _in the description. If the path is not complete i.e. does not lead to a terminal object belonging to a basic type i.e. a bit, a character, a fixnum, a float or a reference, then the non continuable exception `incomplete-path` is raised. The path must not use invalid slot-names, nor symbol _when the type of the corresponding subpart is a basic type which cannot be subparted anymore.

A string can be defined as

```
(deftype Str1 (star character))
```

then some accesses may be defined and used as in

```
(defgetter FirstChar Str1 0)
(FirstChar aStr1)    ;;→the first character of the aStr1 string
(defgetter CharNth Str1 _)
(CharNth sStr1 3)    ;;→the third character of the aStr1 string
```

Defining Form **defsetter**

(`defsetter` *name type-name* { *natural* |*slot-name* |_} ...)

This form defines a writer to a precise subpart of instances of *type-name*. The subpart is described by a path formed by the third and following parameters of the `defsetter` form. The path may be given with numbers, names (if the corresponding subpart was given a label in the `deftype` form) or may also be the special symbol _. In the lattest case the missing index will be given numerically at invokation time. The writer is a function named *name*. That function takes as many arguments as there are _ in the description plus one which is the new value to be written. It is an error to fill a slot with an incorrect value.

Defining Form **defmaker**

(`defmaker` *name type-name*)

This form defines an allocator for instances of *type-name*. The allocator will be named *name* and is a multiple-arity function which takes as many numbers as required to allocate an instance of *type-name*. It is an error not to give to an allocator the right number of arguments or to apply it on arguments which are not naturals.

Predicate **subclass-p**

(`subclass-p` $form_1$ $form_2$)

Type descriptors may be compared with respect to the class hierarchy with this specialized predicate. The result is a boolean and indicates if the value of $form_1$ is a subtype of the value of $form_2$. It is an error to compare objects which are not type descriptors. A $class_1$ is a subclass of $class_2$ if $class_1$ was defined as (`extend` $class_2$ ...). A subclass of a subclass of *class* is a subclass of *class*.

6.3 Predefined Types

Some predefined types exist such as `exception`, `thread`, `function`. They are always visible and need not be imported to be used. Other types might exist and might have been built by `deftype` (except for bootstrap problems) and share the features provided by these forms. These library types may be numerous in an industrial strength Lisp system, but it is not essential within Reasonable Lisp to define them precisely. The sole dotted pair type is given hereafter.

Symbols deserve a special explanation since their use was so widespread in previous lisps. Symbols serve mainly to represent programs and are mainly used by macros. Symbols are predefined but care has been taken not to require them at run-time unless the user needs it.

6.3.1 Dotted Pair

Dotted pair is an extensible type which is defined by

```
(deftype Dotted-Pair (times (label the-car reference)
                            (label the-cdr reference) ))
(defmaker Dotted-Pair-allocator Dotted-Pair)
(defun cons (a d)
  (let ((cell (Dotted-Pair-allocator)))
    (rplaca cell a)
    (rplacd cell d)
    cell ) )
(defun consp (o)
   (type-eq (type-of o) (type Dotted-pair)) )
(defgetter unsafe-car Dotted-Pair the-car)
(defun car (o)
   (if (eq o '())
       '()
       (unsafe-car o) ) )
```

```
(defsetter set-cdr! Dotted-Pair 1) ;1 designates the second field.
(defun rplacd (o d)
  (set-cdr! o d)
  o )
```

In particular one can infer from that description that it is an error to apply `car`, `cdr`, `rplaca` or `rplacd` on an object which does not satisfy `consp`. The old aberration `(eq (car '()) '())` is still true whilst `(consp '())` is still false !

7 Macros

Macros provide a way to extend the syntax of REASONABLE LISP. Macros are suspicious and may induce semantical problems. However their average use is benefitful. To cope with macros only requires to know when and where they are expanded. Given that knowledge one can deal with, even in the case of macros doing kludgy side-effects.

In REASONABLE LISP macros are not first-class objects: they are only functions known as expanders and used in a particular way. The `with-macros` form is a syntactical form which indicates the local use of abbreviations. Macros can also be imported in the importation clause of `defmodule`. Its scope is therefore the entire module body.

Syntax `with-macros`

```
(with-macros ((name_i function-resource)
              ... )
     form )

function-resource ::= ( module-name function-name )
                   |  function
```

This form is pure syntax: it has no value at all. It only extends the set of valid abbreviations as mentionned in the first parameter. Abbreviations may only appear in the second parameter: the body of `with-macros`. A form is an abbreviation if its `car` is a symbol which belongs to the set of valid abbreviations. The first parameter is a list of couples. Each couple specifies an abbreviation trigger and an expander. The expander may be alternately a normal function belonging to another module and of course exported from it, or a functional object. Expanders are immediate citation of functions and are not evaluated. When an abbreviation is recognized, it is expanded i.e. the associated expander is applied on the whole form. It is an error if the expander cannot accept one argument. The result will replace the original form and will be expanded again if it still contains abbreviations. If the expander refers to an unloaded module then this module will be implicitely loaded. To specify an expander which is not exported from the mentionned module will raise, if used, the non continuable `undefined-function` exception.

`with-macros` forms may be arbitrarily nested. Inner syntactic triggers may hide outer ones of the same name.

Every function able to take one argument can be used as a macro. Like every function, expanders are naturally evaluated in the lexical context of the module where they were defined and in the current dynamic context. In particular they may use the global lexical variables or constants of their modules. Expanders and their modules are only useful for expansion, they will not appear in the resulting expansion. Macro expansion takes place at module definition time (see modules). The macroexpansion is performed by the following algorithm

```
(defun macroexpand (form)
  (expand form init-macroenv) )
(defun expand (form env)
  (if (consp form)
     (if (assoc (car form) env)
        (funcall (cassoc (car form) env) form)
```

```
       (if (assoc (car form) special-form-env)
          (funcall (cassoc (car form) env) form env)
          (mapcar (lambda (form) (expand form env))
                  form ) ) )
    form ) )
(constant-define init-macroenv
   (list  ;;all standard macros
     (cons 'dynamic-let
           (lambda (form) ...) )
     ... ) )
(constant-define special-form-env
   (list  ;;all special form walkers
     (cons 'with-macros
           (lambda (form env)
             (expand (third form)
                     (append (mapcar (lambda (pair)
                                       (cons (car pair)
                                             (cond ((functionp (second pair))
                                                    (second pair) )
                                                   (t (find (second pair)
                                                            (module-functions
                                                              (find (first pair)
                                                                    module-env ) ) )) ) ) )
                     env ) ) ) )
     (cons 'quote (lambda (form env)
                     (if (and (consp (cdr form))
                              (null (cddr form)) )
                         form
                         `(error (make-syntax-error)) ) ))
     (cons 'if ... )
     ...
     (cons 'expand (function expand)) ;;!?! the eval of expansion ?!?
     ) )
```

All defining forms and all special forms defined in Reasonable Lisp have an associated code-walker which checks their syntax.

7.1 Examples of Macros

Here is a little example of a module defining a function providing a profiling facility. This function is intended to be imported as a macro

```
(defmodule profiling-list-macros
  (import dotted-pair) ;;imports cons, car, length ...
  (import integer)     ;;imports +, 1+, = ...
  (import format)
  (defun xcons (call)
    (setq xcons-counter (1+ xcons-counter))
    (if (= (length call) 3)
        `(cons ,(caddr call) ,(cadr call))
        (error (make-erroneous-arity)) ) )
  (export xcons)
  (variable-define xcons-counter 0)
  (defun print-xcons-stat ()
    (format t "xcons-counter is ~D." xcons-counter)
    (setq xcons-counter 0) ) )
```

To write more easily macros, one can define the defmacro and macrolet macros. Something like

```
(defmodule macro
    (import dotted-pair)
    (defun defmacro (call)
      `(defun ,(cadr call) (call)
         (let ,(destructure-pattern (caddr call) 'call)
            . ,(cdddr call) ) ) )
    ... )
```

A more richer example is the following adapted from Symbolics documentation:

```
(defmodule collection
   (import ...)
   ;;(with-collection (dotimes (i 5) (collect i))) →(0 1 2 3 4)
   (defun with-collection (call)
     (let ((var (gensym)))
        `(let ((,var '()))
            (with-macros ((collect ,(lambda (call)
                                     ;;push has to be imported in the module
                                     ;;where with-collection is used.
                                     `(push ,(cadr call) ,',var) )))
                  ;;collect will only be expanded here
                  (progn ,@(cdr call)) )
            (nreverse ,var) ) ) )
   )
```

8 Modules

Modules as in other languages provide information hiding and separate definition of non related topics. Modules are linked together by virtue of importation and exportation clauses which determine inter-module visibility on some informations. A module can export types, functions, constants or variables. Names of these objects joined to the name of the module where they are defined allow to retrieve them. Name clashes are avoidable since names can be changed at exportation or importation.

Predefined modules exist with usual functions, constants and variables. These modules are rather tight to allow precise selective linking.

Within a module is a *toplevel*. Toplevel definitions are defined by a grammar which forbids inner definition. Forms which are not toplevel definitions are just set-up code to be executed when the module will be loaded. Only toplevel definitions can be exported. Toplevel definitions may contain setup-code when initializing a constant for example. Toplevel definitions do not have values, they are not evaluated, they only participate to module definition.

Modules can be in two states: *loaded* or *unloaded*. In order to give the user the full control on the order of evaluation, modules may be explicitely loaded by `loadmodule`. They may also be implicitely loaded when an exported function is called. It is the well known *autoload* mode. If calling an exported function loads a module, reading the value of an exported variable or constant is an error, modifying the value of an exported variable is also an error. Nevertheless it is possible to use exported types without loading the module containing their definition.

The important times in the life of a module are

1. Importations are checked syntactically and then all imported references are verified in the modules from which they are exported: constants must exist, be constant and exported, variables must exist, be variables and exported, functions must exist and be exported, types must exist and be exported. If some references are missing the non continuable exception `unknown-reference` will be raised.

   All these references form the initial lexical context (variable, functional and type environment) of the module being defined.

2. The body of the module is then sequentially expanded. All syntactic abbreviations are expanded: atoms stay even while forms may trigger expansion. When a form is expanded, its expansion is reexpanded

until no more abbreviation can be found, its subforms are then expanded from left to right. The syntax of special form is for sure respected.

Initial macros are those which are imported by the `macro` keyword. The set of active macros can be locally extended by `with-macros`.

3. When the body of the module is entirely expanded, all the constants, variables, types and functions defined by toplevel forms are gathered to form the global lexical context of the modules. The global lexical context is made of the global lexical variable environment, the global lexical functional environment and the type environment of the module.

4. The export clauses are then checked. They must only export existing references. It is possible to reexport imported references: modules may also only gather external references.

5. The module environment is then extended to record the freshly defined module. The module gives a location for every constant, variable and function defined in its toplevel. It therefore possible to reference such objects if exported. The module is not yet loaded and all forms that are not defining forms are still not evaluated. It is an error to try to read the value of exported constant or exported variable. It is possible to invoke from outside an exported function, this call involves the automatic load of the module. Nevertheless it is possible to use exported types without loading the module where they were defined.

   All the previous phases are known as *module definition*. Module definition is done in the null lexical and dynamic context. This ends the `defmodule` work and roughly correspond to the construction of an interface to the module.

6. A module can be loaded thanks to `loadmodule`. Moreover a module may be reloaded as much times as wanted. When a module is loaded all its set-up code is sequentially evaluated in the dynamic context of the form `loadmodule`. After being loaded the module is known to be loaded and cannot be any longer implicitely loaded.

Here is a little module which defines some resources

```
(defmodule bof
  (import what-is-necessary)
  (export pi)
  (constant-define pi (compute-pi (dynamic-ref 'number-of-decimals)))
  (dynamic-set '*features*
        (cons 'pi *features*) )
  )
```

When the module will be loaded, the forms

```
(compute-pi (dynamic-ref 'number-of-decimals))
(dynamic-set '*features*
              (cons 'pi (dynamic-ref '*features*)) )
```

will be evaluated in the current dynamic context. Thus

```
(dynamic-let ((number-of-decimals 3))
     (loadmodule bof) )
```

will not define `pi` like

```
(dynamic-let ((number-of-decimals 345))
     (loadmodule bof) )
```

8.1 Modules Related Domains

```
(def-domain "Module" (* ExpTypEnv ExpLexEnv ExpFunEnv ExpExceptEnv
                        TypEnv LexEnv FunEnv ExceptEnv
                        (label initialisation Func)
                        (label state (enum loaded-module
```

```
                                         unloaded-module )) ))

(def-domain "ModulEnv"
  (-> Id (lifted Module modulenv-module-not-found)) )
```

8.2 Module Features

Defining Form **defmodule**

(defmodule *name*
  (*importations...*)
  forms...)

This form defines a module. The module is elaborated according to the previous rules. The module environment is then extended to record this new module: exported resources may then be referenceable. The body of the module i.e. the forms which follow the importation clause must respect the following grammar:

```
<body> ::= <defining-form> . <body>
        |  ( progn <body> ) . <body>
        |  ( <set-up-code> . <body> )
        |  ()
 <defining-form> ::= ( constant-define <name> <set-up-code> )
                  |  ( variable-define <name> <set-up-code> )
                  |  ( defun <name> <variables> <definition> )
                  |  ( defgeneric <name> <variables> )
                  |  ( defmethod <name> <disc-variables> <definition> )
                  |  ( defclass <name> (<super>) <slots> )
                  |  ( defmodule <name> (import . <module-importations>) <body> )
                  |  ( export . <exportations> )
<exportations> ::= <exportation> . <exportations>
                |  ()
<exportation> ::= ( variable <name> )
               |  ( constant <name> )
               |  ( function <name> )
               |  ( type <name> )
               |  ( rename-as <new-name> <exportation> )
<module-importations> ::= <module-importation> . <module-importations>
               |  ()
<module-importation> ::= ( <module-name> . <importations> )
                      |  <module-name>
<importations> ::= <importation> . <importations>
                |  ()
<importation> ::= ( variable <name> )
               |  ( constant <name> )
               |  ( function <name> )
               |  ( type <name> )
               |  ( macro <name> )
               |  ( rename-as <new-name> <importation> )
```

Exportations can only be made on objects defined at the toplevel of the module i.e. within global environments. The mutability of the exported binding is not altered.

Importations can only be made on exported bindings. The kind of binding must be the same between the exported and imported clause.

Special Form **loadmodule**

(`loadmodule` *module-name*)

```
(def (meaning (loadmodule-form name))
  (build :meaning M
    (let ((mod (M.modulenv name)))
      (if (samep mod modulenv-module-not-found)
        (exit :message "Module not found")
        (call :func (module-initialisation mod)
              :values (no-values) ) ) ) ) ) )
```

This special form loads the module named *module-name*. To load the module means that the set-up code identified during the definition of the module is evaluated sequentially. After loading the module will never be implicitely loaded but it can be reloaded: the set-up code will be evaluated again. If the module does not exist then the exception `undefined-module` will be raised.

A module is loaded in the null lexical context.

Special Form **export**

(`export` *exportations*)

The `export` clause allow to export existing resources defined in the toplevel of the current module. the `export` clauses can be written anywhere in the module provided it is in a toplevel place. If a resource does not exist when exportations are processed then the non continuable exception `illegal-export` will be raised. The syntax of *exportations* was defined before and allow to change the name of a resource in a possibly longer name.

8.3 Toplevel of a Module

Two kind of forms may appear in the body of a module. Defining forms or set-up forms. Defining forms defines global resources of the module that may be exported. These global resources are functions, macros, constants or variables: they form together with the imported bindings the initial global environments of the module. They also are what is needed to separately compile (not necessarily efficiently) other modules. Set-up forms are executed every times the module is loaded and usually initialize complex data structures. Once executed, these set-up forms are no longer useful and may be wiped out.

Toplevel forms are determined after full macro expansion of the module.

Defining Form **variable-define**

(`variable-define` *name form*)

Global lexical variable bindings may be created in the toplevel of a module by `variable-define`. The created binding is mutable and the associated value may be altered by `setq`. If a global lexical binding already exists with the same key i.e. the same name (mutable or immutable) then the non continuable exception `lexical-redefinition` will be raised when the enclosing module is elaborated. The *form* will be evaluated when the module will be loaded and its result will be assigned to *name*.

Defining Form **constant-define**

(`constant-define` *name form*)

Global lexical constant bindings may be created in the toplevel of a module by `constant-define`. The created binding is immutable, the associated value may not be altered by `setq`. If a global lexical binding already exists with the same key (mutable or immutable) then the non continuable exception `lexical-redefinition` will be raised when the enclosing module is elaborated. The *form* will be evaluated when the module will be loaded and its result will be assigned to *name*.

8.4 Applications

Modules allow to define applications i.e. a whole set of functions and other things, to perform some computation. These applications may be initiated by the underlying operating system and may also be submitted some parameters (the well known options).

A module defines as many applications as there are exported functions defined in it. When the underlying operating system initiates an application, the starting function is applied on a set of parameters submitted by the operating system. These parameters are implementation dependent. It is useful to have multiple argument functions as starting point in order to minimize operating system dependence. For example, in UNIX[5] an entry point may look like

```
(multiple-arity-lambda arg
   (let ((argc (mv-ref arg 0))
         (argv (mv-ref arg 1)) )
      ( application) ) )
```

If one admits the pseudo special form *startmodule*, the denotation looks like

```
(def (module-meaning (startmodule module-name function-name arguments))
  (build :module-meaning M
    (let ((mod (M.modulenv module-name)))
      (if (samep mod modulenv-module-not-found)
        (exit :message "Module not found")
        (let ((altloc ((module-expfunenv mod) function-name)))
           (if (samep altloc funenv-altloc-not-found)
              (exit :message "Entry Point not found")
              (call :func ((module-funenv mod) altloc)
                    :values arguments
                    :schenv init.schenv
                    :modulenv M.modulenv
                    :lexenv (module-lexenv mod)
                    :multlexenv init.multlexenv
                    :funenv (module-funenv mod)
                    :dynenv init.dynenv
                    :dynesc init.dynesc
                    :lexesc init.lexesc
                    :store init.store
                    :altstore init.altstore
                    :handlerenv init.handlerenv
                    :cont (build :cont C
                            (exit :message "End of application") ) ) ) ) ) ) ) )
```

A program in REASONABLE LISP is a collection of modules i.e. a sequence of `defmodule` expressions. No other form is allowed. Module definitions may be submitted one by one or by group. The resulting effect on the module environment depends on that grouping.

Some modules may be gathered into a module which only reexports some part of its importations. For example,

```
(defmodule gather
  (import (A bar) (B hux))
  (export bar) )
```

[5] UNIX is a trademark of AT&T.

9 Functions

Functions are the means to express algorithms. Functions have variables which will be bound in some environment when the function is applied. Different kinds of functions exist: they may have a fixed or an indefinite number of arguments. They can also be generic in which case when applied a precise method will be found after the types of the arguments. In this description, `lambda` is not a special form but a macro which allows to define numerous *lambda-list* keywords.

Since REASONABLE LISP is a **Lisp**$_2$, the name of a functional application is looked for in the lexical functional environment. On the other hand parameters are handled by the normal evaluation mechanism. We will refer to the two mechanisms with the words "functional evaluation" and "parametric evaluation". Functional evaluation is automatically invoked on part in functional position while the parametric evaluation is applied on parts which are in parametric positions. To permute the default behaviours exist the `function` special form which converts a function name into the associated first-class function, the converse is done by the function `funcall`.

9.1 Function Related Domains

```
(def-domain "Func"
  (-> Values Dynenv DynEsc HandlerEnv AltStore Store Cont Answer) )
```

9.2 Function General Features

no **`lambda`**

The name `lambda` is free in order to be usable as a macro with all needed or unneeded features. Ask your local guru where to find the appropriate library. Here `lambda` with a symbol variable-list expands to `multiple-arity-lambda` and, with a real variable-list expands to `fixed-arity-lambda`.

Special Form **`function`**

(`function` *function-name*)

```
(def-meaning (functional-reference name)
  (build :meaning M
    (let ((loc (M.funenv name)))
      (if (samep loc FunEnv-location-not-found)
        (exit :message "Undefined Function")
        (call :cont M.cont
              :values (Values<-Value (Value<-Func (M.store loc)))) ) ) ) )
```

The `function` special form evaluates its parameter by the functional evaluator and returns as value the associated function. If no function is associated to the name *function-name* in the lexical functional environment then the exception `function-unfound` will be raised.

`function` may be abbreviated by `#'` thus `#'cons` is the same as `(function cons)`.

Syntax ***functional application***

(*function-name parameters...[keyworded-parameters...]*)

```
(def-meaning (application function arguments)
  (build :meaning M
    (call :meaning (meaning function)
          :cont (build :cont C
                  (call :meaning (meaning arguments)
                        :cont (build :cont NC
                                (call :Func (Func<-Value (first C.values))
```

```
                                          :values NC.values ) ) ) ) ) ) )
```

An application is computed after the following rules:

1. The functional binding associated to the name which is in `car` of the form is looked for in the functional environment which ends up into the global functional environment of the module where appears the original form. If no binding exists the the non continuable exception `function-unfound` is raised.

2. The parameters are sequentially evaluated from left to right and multiple values are coerced into single values. When a keyword is encountered all remaining parameters must be keyworded and are evaluated from left to right skipping the keywords that are pure syntax. Parameters before keywords are passed positionally. Parameters after keywords may adopt whatever order wanted provided that eventually all arguments appear and no argument is given twice (or more).

3. The function is then applied on its arguments. It is an error not to provide a function with the right number of arguments (see `arity-p`). Predefined functions try to perform necessary domain membership tests before any other computations.

Function funcall

(`funcall` *function parameters...*)

```
(def-initial-function funcall
  (build :Func F
    (if (>= (*length F.values) 1)
       (tagcase (first F.values)
         (Func (call :func (Func<-Value (first F.values))
                     :values (*tail F.values) ))
         (t (exit :message "Wrong argument: not a function")) )
       (exit :message "Erroneous Number of Arguments") ) ) )
```

Like every function, all parameters of `funcall` are evaluated by the parametric evaluator. The first argument must be a function i.e. an object which answers true under `functionp`, which is then applied on the other arguments. `funcall` returns what returns the application and may thus return multiple values. If keywords appear in a `funcall` form then they are intended to be used by `funcall` and not by the first argument: it is not possible to make a keyworded call to a computed function. The same rule applies to `apply`.

Predicate functionp

(`functionp` *form*)

```
(def-initial-function functionp
  (build :Func F
     (if (= (*length F.values) 1)
       (tagcase (first F.values)
         (Proc (call :cont F.cont
                     :values (Values<-Value (Value<-boolean boolean-true)) ))
         (t (call :cont F.cont
                  :values (Values<-Value (Value<-Boolean boolean-false)) )) )
       (exit :message "Erroneous Number of Arguments") ) ) )
```

returns true only if its argument is a function. Functions are atomic objects which are opaque, it is not possible to retrieve the external representation of their body. The only property of functions which can be asked for is wether they can accept a given number of arguments.

Predicate **arity-p**

(`arity-p` *function number*)

answers true only if the function value of the first parameter can be applied on *number* arguments without raising the `incorrect-arity` exception. It is an error not to provide `arity-p` with a function and an integer.

Defining Form **defun**

(`defun` *name variable-s form...*)

The `defun` form defines a function named *name* in the current global functional environment i.e. in the current module. `defun` is a defining form and can only appear in the toplevel of a module. the *variable-s* parameter indicates whether is defined a fixed arity function (if it is a list of symbols) or a multiple arity function (if it is a single symbol). If a function of the same name exists in the current functional environment (wether defined before or imported) then the non continuable `function-redefinition` exception will be raised.

Special Form **flet**

(`flet` (($name_i$ ...)
   ...)
  form...)

```
(def (meaning (flet-form names functions body))
  (build :meaning M
    (let ((frame (new-altlocations (*length names) M.altstore)))
       (call :meaning (meaning functions)
             :cont (build :cont C
                     (call :meaning (meaning body)
                           :funenv (extend* M.funenv names frame)
                           :altstore (extend C.altstore frame C.values)
                           :cont M.cont ) ) ) ) ) )
```

Special Form **labels**

(`labels` (($name_i$ ...)
   ...)
  form...)

```
(def (meaning (labels-form names functions body))
  (build :meaning M
    (let ((frame (new-altlocations (*length names) M.altstore)))
       (call :meaning (meaning functions)
             :funenv (extend* M.funenv names frame)
             :cont (build :cont C
                     (call :meaning (meaning body)
                           :altstore (extend C.altstore frame C.values)
                           :cont M.cont ) ) ) ) ) )
```

9.3 Fixed Arity Functions

Special Form **fixed-arity-lambda**

(`fixed-arity-lambda` (*variables*)
  forms...)

```
(def-meaning (fixed-arity-abstraction variables body)
```

```
  (build :meaning M
     (let ((fn (build :Func F
                  (if (= (*length F.values) (*length variables))
                      (let ((frame (new-locations F.store (*length variables))) )
                         (call :meaning (meaning body)
                               :lexenv (extend* M.lexenv variables frame)
                               :store  (extend* F.store frame F.values) ) )
                      (exit :message "Erroneous number of arguments") ) )))
        (call :cont M.cont
              :values (values<-value (Value<-Func fn)) ) ) ) )
```

Functions with a fixed arity are constructed with `fixed-arity-lambda`. The created function closes all variables that appear free in its body. More precisely an abstraction closes the lexical variable environment, the lexical multiple value environment, the lexical functional environment, the lexical escape environment and the tagged escape environment. When applied its parameters will be evaluated in the current context, then will be bound to the *variables* in the lexical variable environment. Finally the *forms...* which compose the body of the function will be sequentially evaluated in the extended lexical variable environment. The created bindings are mutable and may be altered by `setq`.

Syntax *variable*

variable

```
(def-meaning (reference name)
  (build :meaning M
    (let ((loc (M.lexenv name)))
      (if (samep loc LexEnv-location-not-found)
        (exit :message "Undefined Identifier")
        (call :cont M.cont
              :values (Values<-Value (M.store loc))) ) ) ) )
```

The value of a variable of a `fixed-arity-lambda` may be retrieved when citing the variable. The value is looked for in the lexical variable environment which ends up in the global lexical variable environment of the module where appears the text of the `fixed-arity-lambda` form. If the variable does not have a lexical binding then the continuable exception `lexical-variable-unfound` is raised.

Special Form `setq`

(`setq` *variable form*)

```
(def-meaning (assignment name form)
  (build :meaning M
    (call :meaning (meaning form)
      :cont (build :cont C
              (if (= (*length C.values) 0)
                (exit :message "Empty multiple values")
                (let ((val (first C.values))
                      (loc (M.lexenv name)) )
                  (if (samep loc LexEnv-location-not-found)
                    (wrong :message "Undefined Identifier"
                           :values name )
                    (call :cont M.cont
                          :values (values<-value val)
                          :store (extend C.store loc val) ) ) ) ) ) ) ) )
```

First the second parameter is evaluated in the current context. The lexical binding of the variable *name* is looked for in the current lexical variable environment which ends up in the global lexical variable environment of the module where appears the `setq` form. If no binding is found then the non continuable exception `lexical-variable-unfound` will be raised. If the binding is found then it is altered to associate the value of the second parameter to *name*. The result of the `setq` form is the value of the second parameter.

9.4 Generic Functions

Generic functions are composed of a set of methods. Generic functions are first-class mutable objects to which methods can be added. Once a method is added it is not possible to remove it. Methods can only be added thanks to the defining form `defmethod`. The added method must be congruent to the definition of the generic which must be performed before.

Defining Form `defgeneric`

(`defgeneric` *name* (*variables*))

The list of variables have a structure similar to those of `fixed-arity-lambda`. This defining form defines a generic function named *name*. When applied at that time, this generic function will raise the continuable exception `method-unfound`.

Defining Form `defmethod`

(`defmethod` *name* (*variables*)
form...)

`defmethod` adds a method to the generic function *name*. If this generic function does not exist, the non continuable exception `generic-unfound` will be raised. If the generic function exists, its variable-list is compared to that of the method. If non congruent, the non continuable exception `non-congruent-method` is raised. The method is congruent to the variable-list of the generic function if the method can accept all arguments that may be submitted to the generic function. It is an error if the method can accept more than the generic.

Once created a generic function grows as more and more methods are added. If invoked while macroexpansion, the generic will be used as it is i.e. if the module where it was defined is not loaded, then this module is loaded and the generic will be used with its current methods. No attempt will be made to load all the modules that may add a method to it. It is the user responsability to load all the needed modules.

9.5 Multiple Arity Functions

Functions created by `multiple-arity-lambda` can take as many arguments as provided by functional application. All these arguments are gathered in a multiple-value entity which can only be accessed or modified via special forms.

Special Form `multiple-arity-lambda`

(`multiple-arity-lambda` *name forms*...)

```
(def-meaning (multiple-arity-abstraction variable body)
  (build :meaning M
     (let ((fn (build :func F
                      (let ((frame (new-locations F.store (*length F.values))))
                        (call :meaning (meaning body)
                              :lexmultenv (extend M.lexmultenv
                                                  variable frame )
                              :store (extend* F.store frame F.values) ) ) )))
       (call :cont M.cont
             :values (values<-Value (Value<-Func fn)) ) ) ) )
```

returns a function which when applied will bind *name* to the multiple value object in the lexical multiple value environment and evaluate the *forms* in this environment. The binding is immutable, has indefinite extent and lexical scope. The associated multiple value entity is mutable. The created function captures all the lexical context of its creation.

Besides creating such functions, some forms are provided to manipulate them along with multiple values.

Multiple argument functions can be applied with the default syntax for function application

(*multiple-arity-function single-argument* ...)

where every parameter are evaluated yielding a single value that are collected to be submitted to the function. For example,

```
(funcall (multiple-arity-lambda args (1- (mv-length args)))
         1 2 3 4 5 ) → 5
```

Special Form mv-call

One can also gather multiple values by

(`mv-call` *function mv-argument* ...)

```
(def-meaning (multiple-application function arguments)
   (build :meaning M
      (call :meaning (meaning function)
            :cont (build :cont C1
                     (if (= (*length C1.values) 0)
                       (exit :message "Empty multiple values")
                       (tagcase (first C1.values)
                          (Func (call :meaning (meaning arguments)
                                      :cont (build :cont C2
                                               (call :func (func<-Value (first C1.values))
                                                     :values C2.values
                                                     :cont M.cont ) ) ))
                          (t (exit :message "Not a function")) ) ) ) ) ) )

(def-meaning (mv-parameters first rest)
  (build :meaning M
    (call :meaning (meaning first)
          :cont (build :cont C1
                   (call :meaning (meaning rest)
                         :cont (build :cont C2
                                  (call :cont M.cont
                                        :values (*append C1.values
                                                         C2.values ) ) ) ) ) ) ) )
```

The first parameter of `mv-call` is evaluated and must yield a function otherwise the exception `not-applyable` is raised. The first parameter is evaluated in a parametric place, not as in a functional place. The other parameters are then evaluated sequentially, their values (even multiple) are collected and submitted to the function of first parameter.

Fixed arity functions may also be invoked by `mv-call`

```
(mv-call (function cons) 1 2)          → (1 . 2)
(mv-call (function cons) (values 1 2)) → (1 . 2)
```

Special Form mv-length

Multiple values can be inspected for their length by `mv-length`

(`mv-length` *name*)

```
(def-meaning (multiple-value-length label)
  (build :meaning M
```

```
    (let ((locs (M.lexmultenv label)))
      (if (samep locs lexmultenv-locations-not-found)
        (exit :message "No multiple argument")
        (call :cont M.cont
              :values (Values<-Value
                        (Value<-Num
                         (make-num :internal-index (*length locs)) ) ) ) ) ) ) )
```

The first parameter must be the name of a multiple value otherwise the exception `unknown-multiple-value` is raised. `mv-length` returns the number of values composing this multiple value.

Special Form **mv-ref**

A single value may be extracted from a multiple value by

(mv-ref *name index*)

```
(def-meaning (multiple-value-ref label form1)
  (build :meaning M
    (let ((locs (M.lexmultenv label)))
      (if (samep locs lexmultenv-locations-not-found)
        (exit :message "No multiple argument")
        (call :meaning (meaning form1)
              :cont (build :cont C
                        (if (= (*length C.values) 0)
                          (exit :message "Empty multiple values")
                          (tagcase (first C.values)
                              (Num (let ((n (natural-term-internal-index
                                               (Num<-Value (first C.values)) )))
                                     (if (and (<= 0 n) (< n (*length locs)))
                                       (call :cont M.cont
                                             :values
                                               (values<-Value (C.store (*elt locs n))) )
                                       (wrong :message "Out of range index"
                                              :values (values<-value (first C.values)) ) ) ))
                             (t (exit :message "Not a number")) ) ) ) ) ) ) ) )
```

The first parameter must be the name of a multiple value otherwise the exception `unknown-multiple-value` is raised. The second parameter is evaluated and must yield an *index* to the multiple value $0 \leq index <$ *number of multiple values*. It is an error to provide an illegal index. `mv-ref` returns the $index^{th}$ value of the multiple value. Values inside a multiple value are numbered from zero.

Special Form **mv-set**

A single value may be modified within a multiple value by

(mv-set *name index value*)

```
(def-meaning (multiple-value-set label form1 form2)
  (build :meaning M
    (let ((locs (M.lexmultenv label)))
      (if (samep locs lexmultenv-locations-not-found)
        (exit :message "No multiple argument")
        (call :meaning (meaning form1)
              :cont (build :cont C1
                        (if (= (*length C1.values) 0)
                          (exit :message "Empty multiple values")
                          (call :meaning (meaning form2)
```

```
                                  :cont (build :cont C2
                                          (if (= (*length C2.values) 0)
                                            (exit :message "Empty multiple values")
                                            (tagcase (first C1.values)
                                                (Num
                                                 (let ((n (natural-term-internal-index
                                                            (Num<-Value (first C1.values)))))
                                                   (if (and (<= 0 n) (< n (*length locs)))
                                                     (call :cont M.cont
                                                           :store (extend C2.store
                                                                          (*elt locs n)
                                                                          (first C2.values) )
                                                           :values
                                                           (values<-Value (first C2.values)) )
                                                     (wrong :message "Out of range index"
                                                            :values (values<-value
                                                                      (first C1.values)) ) ) ))
                                               (t (exit :message "Not a number")) ) ) ) ) ) ) ) ) ) )
```

The first parameter must be the name of a multiple value otherwise the exception `unknown-multiple-value` is raised. The second parameter is evaluated and must yield an *index* to the multiple value $0 \leq index <$ *number of multiple values*. It is an error to provide an illegal index. The third parameter is evaluated and will replace the $index^{th}$ value of the multiple value. `mv-set` returns that very value.

Function `values`

Multiple values are produced by the functions `values` and `make-values`

(`values` *forms...*)

```
(def-initial-function values
  (build :Func F
    (call :cont F.cont
          :values F.values ) ) )
```

`values` evaluates sequentially each of its parameters and return the whole set of these values. If there is a pending `mv-call` then these values will be aggregated otherwise if the current continuation of `values` do not expect multiple values then this multiple value is coerced into its first one (or NIL if the multiple value is void).

Function `make-values`

Another form can produce a multiple value

(`make-values` *number* [*initial-value*])

This form evaluates its first parameter which must yield a number satisfying $number \geq 0$. The second parameter is evaluated and will be used to produce a multiple value with *number* slots each initialized to the value of the second parameter. If the first parameter does not yield a correct number, the exception `negative-multiple-values` will be raised. The function `values` can be explained in terms of `make-values` but may be implemented in a faster way.

Parts of multiple value can be submitted as multiple value thus realizing a kind of coercion between a multiple value and a multiple argument.

Special Form `mv-values`

(`mv-values` *name* [*start* [*end*]])

```
(def-meaning (multiple-value-send label)
  (build :meaning M
```

```
    (let ((locs (M.lexmultenv label)))
      (if (samep locs lexmultenv-locations-not-found)
        (exit :message "No multiple argument")
        (call :cont M.cont
              :values (send-values M.store locs) ) ) ) ) )

(define (send-values store locs)
  (if (*null locs)
    (no-values)
    (*insert (store (*head locs)) (send-values store (*tail locs))) ) )
```

This form extracts from the multiple value named *name* the specified slice between *start* (included) and *end* (excluded). If *end* is not specified then all values after the $start^{th}$ will be used. If *start* is not specified then *start* is considered to be zero. Values inside a multiple value are numbered from zero to `(1- (mv-length` *name*`))`.

10 Dynamic Bindings

Dynamic variables are not really variables but a way to link informations to symbols with a dynamic extent and an indefinite scope. Dynamic variables are strictly separated from lexical variables, they are handled by a different syntax and a different semantics. Accesses to dynamic variables are encapsulated by a set of specialized special forms.

REASONABLE LISP extends the concept of dynamic variables and is able to link any information to any other. The first information is therefore not restricted to be a symbol. This behaviour is similar to `catch` which is able to bind a continuation to any other object.

There is no global dynamic environment. Dynamic bindings must always be established by the special form `dynamic-bind` and will be destroyed on exit.

10.1 Dynamic Variable Related Domains

```
(def-domain "Dynenv"
  (-> Value (lifted Location dynenv-location-not-found)) )
```

10.2 Dynamic Variable Features

Special Form **`dynamic-bind`**

(dynamic-bind ((*source-form*$_i$ *target-form*$_i$)
...)
form...)

```
(def-meaning (dynamic-binding form1 form2 body)
  (build :meaning M
    (call :meaning (meaning form1)
          :cont (build :cont C1
                  (if (= (*length C1.values) 0)
                    (exit :message "Empty multiple values")
                    (call :meaning (meaning form2)
                          :cont (build :cont C2
                                  (if (= (*length C2.values) 0)
                                    (exit :message "Empty multiple values")
                                    (call :meaning (meaning body)
```

```
                                        :dynenv (extend M.dynenv
                                                        (first C1.values)
                                                        (new-location C2.store) )
                                        :store (extend C2.store
                                                       (new-location C2.store)
                                                       (first C2.values) )
                                        :cont M.cont ) ) ) ) ) ) ) ) )
```

The first parameter is a list of bindings to establish. All forms are sequentially evaluated in the current context and then bound. The body of the `dynamic-bind` form is then evaluated in the extended dynamic environment. The `dynamic-bind` form returns what returns its body. When the `dynamic-bind` form returns, the bindings are suppressed. Dynamic bindings cannot be closed. Dynamic bindings are mutable.

Special Form `dynamic-ref`

`(dynamic-ref` *form*`)`

```
(def-meaning (dynamic-referencing form)
  (build :meaning M
    (call :meaning (meaning form)
          :cont (build :cont C
                  (if (= (*length C.values) 0)
                    (exit :message "Empty multiple values") ;;un peu dur !
                    (let ((loc (M.dynenv (first C.values))))
                      (if (samep loc dynenv-location-not-found)
                        (wrong :message "Unbound dynamic value"
                               :values (Values<-Value (first C.values)) )
                        (call :cont M.cont
                              :values (Values<-Value (C.store loc)) ) ) ) ) ) ) ) )
```

This form evaluates its first parameter and returns its associated value in the dynamic environment. If no binding exists for the value of the first parameter, the continuable exception `dynamic-binding-unfound` is raised.

Special Form `dynamic-set`

`(dynamic-set` *source-form target-form*`)`

```
(def-meaning (dynamic-alteration form1 form2)
  (build :meaning M
    (call :meaning (meaning form1)
          :cont (build :cont C1
                  (if (= (*length C1.values) 0)
                    (exit :message "Empty multiple values")
                    (call :meaning (meaning form2)
                          :cont
                           (letrec ((c (build :cont C2
                                         (if (= (*length C2.values) 0)
                                           (exit :message "Empty multiple values")
                                           (let ((loc (M.dynenv (first C1.values))))
                                             (if (samep loc dynenv-location-not-found)
                                               (wrong :message "Unbound dynamic value"
                                                      :values
                                                       (Values<-Value
                                                        (first C1.values) )
                                                      :cont c )
```

```
                                    (call :cont M.cont
                                          :values
                                           (Values<-Value
                                            (first C2.values) )
                                          :store (extend C2.store
                                                         loc
                                                         (first C2.values) )
                                          ) ) ) ) ) ))
                            c ) ) ) ) ) ) )
```

This form allow to modify a value in the dynamic environment. First the *source-form* is evaluated, then its corresponding binding is looked for in the dynamic environment. If no binding exists for that value then the non continuable exception `dynamic-binding-unfound` is raised. If the binding exists, the associated value is modified to become the value of *target-form*. The result of the assignment is that very value.

Macro **`dynamic-let`**

(dynamic-let ((*variable$_i$ form$_i$*) ...)
    form...)
    is
(dynamic-bind (((quote *variable$_i$*) *form$_i$*) ...)
    form...)

Here follows a little example with the previous forms

```
(dynamic-bind (('*print-pretty* t))
   ...
      (dynamic-set '*print-pretty* (not (dynamic-ref '*print-pretty*)))
      ...
   (dynamic-bind (( (car pair) (cdr pair) )
                  ( (cdr pair) (car pair) ) )
        ... ) )
```

11 Dynamic Extent Objects

Dynamic extent objects such as dynamic escapes or dynamic bindings may be explained with the `stack-let` special form.

Special Form **`stack-let`**

(stack-let ((*name$_i$ allocation-form$_i$*) ...)
    form...)

allocation-form can only be forms where appear constructors in functional position. Constructors are predefined contructors or functions defined by `defmaker`. `stack-let` allocates all the necessary objects and confer them a dynamic extent. These objects may be used without restriction in the body of the `stack-let` form. They are first-class objects and can be given or returned as value. When exiting the `stack-let` form wether naturally or by an escaping form, they disappear. It is thus an error to refer to a dynamic extent object outside its extent.

12 Usual Special Forms

These are the usual ones `progn`, `if` and `quote`.

Special Form **`progn`**

(progn *form*...)

```
(def-meaning (binarySeq form1 form2)
  (build :meaning M
     (call :meaning (meaning form1)
           :cont (build :cont C
                   (call :meaning (meaning form2)
                         :cont M.cont ) ) ) ) )
```

All the *form...* are sequentially evaluated from left to right. The `progn` form returns what returns the last form of its body. There are many places where forms are sequentially evaluated as if in a `progn` form: these are known as being in an implicit `progn`. The last forms of much special forms (`fixed-arity-lambda`, `catch`, ..) are considered to be in an implicit `progn`.

Special Form `if`

(`if` *condition consequent alternative*)

```
(def-meaning (alternative cond then else)
  (build :meaning M
     (call :meaning (meaning cond)
         :cont (build :cont C
                  (if (= (*length C.values) 0)
                     (exit :message "Empty multiple values")
                     (tagcase (first C.values)
                       (Boolean (if (samep boolean-false
                                        (Boolean<-Value (first C.values)) )
                                  (call :meaning (meaning else))
                                  (call :meaning (meaning then)) ))
                       (t (call :meaning (meaning then))) ) ) ) ) ) )
```

The conditional first evaluates its first parameter. If its value is true i.e. not false, then the `if` form returns what returns the *consequent* part otherwise it returns what returns the *alternative* part.

Special Form `quote`

(`quote` *data*)

```
(def-meaning (quotation data)
  (build :meaning M
      (call :cont M.cont
            :values (Values<-Value (Value<-Data data)) )) )
```

Data can appear within programs in two forms: directly if no syntactical confusion can arise, or within `quote` forms. Data that are both atomic and readable[6] are legal parameters of `quote`. So are symbols, numbers or characters. Threads, functions are illegal since they do not have a readable syntax. Compound data such as list or vectors can also be quoted but implies a reconstruction of the compound data. Thus the meaning of `(quote (a b c))` is the same as `(list (quote a) (quote b) (quote c))` but pay attention to the possible capture of the `list` name.

Unquoted data such as numbers or strings hava the same meaning as if they were quoted. This also exclude threads or functions. Strings are so used that they have a special syntax: `"foo"` really means `(string #\f #\o #\o)`.

Since embedded data are now reconstructed again and again as programs need them, it is probably better to share them with `let` or to define them globally in a module. Mutations on such reconstruction is therefore deterministic.

[6] A data is readable (by REASONABLE LISP) if the user can write it.

13 Continuations

Some special forms allow programs to grab and store their continuation while different special forms may retrieve and invoke them. *Escape binding forms* (`block`, `tagbody` and `catch`) associate their continuation with a key in the appropriate environment, see figure 1, and then process their body. The key is usually an identifier (as in `block` or `tagbody`) but sometimes may also be an unrestricted object (as in `catch`). *Escaping forms* (`return-from`, `go` and `throw`) allow to invoke continuations, the associated binding form is then said to be *escaped from*. Escaping forms retrieve the appropriate continuation from their associated environment thanks to the key.

binding form	escaping form	associated environment
`block`	`return-from`	lexical escape environment
`tagbody`	`go`	tagged escape environment
`catch`	`throw`	dynamic escape environment

Table 1: Continuation handling special forms

Except for `go`, escaping forms take value(s) which become the value(s) returned by the escaped binding form. It is thus possible to return multiple values via `return-from` or `throw`. These multiple values will naturally be coerced into single value if the grabbed continuation expect a single value.

Continuations have a dynamic extent: a continuation may only be invoked when *active* i.e. during the computation of the body of the binding form. When a binding form is normally exited or escaped from, the grabbed continuation becomes obsolete and cannot be used. Alternately said, continuations may at most be invoked once and typically are stack-allocated for implementations which use a stack. The bindings established by `block` and `tagbody` have a lexical scope within their body whilst `catch` establishes a binding with a dynamic scope.

Another special form is strongly connected to escapes: `unwind-protect` which allow a program to evaluate a form under the protection of some clean-up forms. This clean-up code is ensured to always be evaluated when the protected form is normally exited or even escaped from, whatever the escaping form is. `unwind-protect` is viewed as a special form which extends all the active continuations in order to ensure that the clean-up code will be evaluated.

13.1 Escapes Related Domains

Lexical and tagged escape environments are closed under abstraction creation; conversely the dynamic escape environment appears in the arguments of abstraction denotation.

Escape binding forms extend their associated environment and evaluate their body in this extended environment. They normally return what returns the last form of their body and thus can transmit multiple values.

Escaping forms look for a key in the appropriate environment, normally get an index, look for an active continuation associated to that index and finally apply it on the value(s) to be transmitted.

```
(def-domain "DynEsc"
  (-> Value (lifted AltLocation dynesc-AltLocation-not-found)) )
(def-domain "AltStore"
  (newable AltLocation (-> AltLocation (lifted AltValue AltStore-Altvalue-not-found))) )
(def-domain "AltValue" (+ Cont      ;;for block, catch
                          Monitor   ;;for schedulers
                          ) )
(def-domain "LexEsc"
  (-> Id (lifted AltLocation lexesc-AltLocation-not-found)) )
```

13.2 Escape Features

Special Form **block**

(block *block-name forms...*)

```
(def-meaning (lexical-block label body)
  (build :meaning M
    (let* ((altloc (new-altlocation M.altstore))
           (c (build :cont C
                (call :cont M.cont
                      :altstore (extend C.altstore altloc
                                        altstore-altvalue-not-found ) ) )) )
      (call :meaning (meaning body)
            :lexesc (extend M.lexesc label altloc)
            :altstore (extend M.altstore altloc (AltValue<-Cont c))
            :cont c ) ) ) )
```

The first parameter is not evaluated and must be an identifier which will be associated to the continuation of the `block` form in the lexical escape environment. The created binding is lexical i.e. is only visible within the body of `block`. The associated continuation has a dynamic extent and may be invoked only during the evaluation of the body of `block`.

The other parameters form the body of the `block` form and are considered to be grouped into an implicit `progn`, the denotation thus considers a single form for the body.

The `block` form normally returns whatever returns its body. The `block` form may be prematurely escaped from if a `(return-from` *block-name final-form*`)` is evaluated. Since the binding between the *block-name* and the continuation is lexical, `return-from` forms may only appear in the body of `block`. This binding may be captured by closures but must still be used during the evaluation of the body of `block` i.e. in the dynamic extent of the `block` form.

Special Form **return-from**

(return-from *block-name form*)

`return-from` is the escape form related to the `block` binding form.

```
(def-meaning (lexical-escape label form)
  (build :meaning M
    (let ((altloc (M.lexesc label)))
      (if (samep altloc lexesc-altlocation-not-found)
        (exit :message "Undefined continuation")
        (call :meaning (meaning form)
              :cont (build :cont C
                      (if (samep (M.altstore altloc)
                                 altstore-altvalue-not-found )
                        (wrong :message "Obsolete continuation"
                               :values nil )
                        (call :cont (Cont<-AltValue (M.altstore altloc))
                              :values C.values ) ) ) ) ) ) ) )
```

The first parameter is not evaluated and must be an identifier. This identifier must be the same as one of the names used by lexically enclosing `block` forms. If it is not the case the exception `unknown-return-from` is raised with the current continuation as resumption continuation of the exception.

The second parameter is then evaluated in the current lexical and dynamic context. After that evaluation (and if not already escaped for other reason), the continuation of the appropriate enclosing `block`, as previously determined, is checked for obsolescence. If it is obsolete then the exception `obsolete-block` is raised with the current continuation as resumption continuation. In the other (normal) case, the continuation is still active and is therefore applied on the value(s) yielded by the second parameter.

Macro **tagbody**

`tagbody` introduces a sequence of expressions which may be optionally labelled by symbols or numbers. `tagbody` allow the (in)famous `go` form which meaning is to escape the current context and branch to the expression following the intended label of the `tagbody`. `tagbody` is a macro that uses the lexical escapes and their associated forms: `block` and `return-from`.

```
(tagbody π0* {αi πi*})
```

is

```
(block EXIT        ;;I hope it is correct !
  (let ((LABEL (function G0)))
     (with-macros ((go ,(lambda (call)
                          `(throw GOTO
                             ,(case (second call)
                               (α1 `(function G1))
                               (αi `(function Gi))
                               ...
                               (t `(error (make-tagbody-label-unfound))) ) )))
     (labels ((G0 (lambda () π0* (G1)))
              (Gi (lambda () πi* (Gi+1)))
              ...
              (Gn (lambda () πn* (return-from EXIT nil))) )
       (loop
         (setq LABEL (catch GOTO (funcall LABEL))) ) ) ) ) )
```

where *LABEL*, *GOTO* and *EXIT* are gensym'ed labels. The expansion is quite complicated since the dynamic context is escaped from when a `go` is performed.

Macro **go**

See above. Note that to have `tagbody` and `go` as macros is legal since special forms may be implemented as macros (and conversely). Since all used labels are generated, there is no possible conflict nor power reduction.

Special Form **catch**

(`catch` *tag forms* ...)

```
(def-meaning (dynamic-block form body)
  (build :meaning M
     (call :meaning (meaning form)
           :cont (build :cont C1
                   (if (= (*length C1.values) 0)
                     (exit :message "Empty multiple values")
                     (let* ((altloc (new-altlocation M.altstore))
                            (c (build :cont C2
                                 (call :cont M.cont
                                       :altstore (extend C2.altstore altloc
                                                   altstore-altvalue-not-found ) ) )))
                       (call :meaning (meaning body)
                             :dynesc (extend M.dynesc (first C1.values) altloc)
                             :altstore (extend M.altstore altloc
                                                 (AltValue<-Cont c) )
                             :cont c ) ) ) ) ) ) )
```

The first parameter is evaluated, `catch` then binds its result to its current continuation in the dynamic escape environment. The created binding is dynamic and therefore visible from everywhere. The associated continuation has a dynamic extent and may be invoked only during the evaluation of the body of `catch`.

The other parameters form the body of the `catch` form and are considered to be grouped into an implicit `catch`, the denotation thus considers a single form for the body.

The `catch` form normally returns whatever returns its body. The `catch` form may be prematurely escaped from if a (`throw` *tag final-form*) is evaluated where the value of the *tag* is `eq` to the value of the *tag* of the initial `catch`. The dynamic escape environment cannot be closed.

Special Form **throw**

(`throw` *tag form*)

```
(def-meaning (dynamic-escape form body)
  (build :meaning M
    (call :meaning (meaning form)
          :cont (build :cont C
                  (if (= (*length C.values) 0)
                   (exit :message "Empty multiple values")
                   (let ((altloc (M.dynesc (first C.values))))
                     (if (samep altloc dynesc-altlocation-not-found)
                       (wrong :message "Absent continuation"
                              :values (Values<-Value (first C.values)) )
                       (call :meaning (meaning body)
                             :cont (Cont<-AltValue (M.altstore altloc)) ) ) ) ) ) ) ) )
```

The first parameter is evaluated and yield the throw-tag. This throw-tag is looked for in the dynamic escape environment and the exception `unknown-throw` is raised if unfound. The resumption continuation associated to that exception is that of the `throw` form.

The second parameter is then evaluated in the current lexical and dynamic contexts and the result(s) are submitted to the previously determined continuation.

Special Form **unwind-protect**

(`unwind-protect` *form cleanup-forms...*)

```
(def-meaning (protection form1 form2)
  (build :meaning M
    (let ((wrapper (lambda (k)
              (build :cont C1
                 (call :meaning (meaning form2)
                       :cont
                       (build :cont C2
                          (call :cont k
                                :values C1.values ) )
                       :altstore
                       (altstore-lambda (altloc)
                          (let ((altvalue (C1.altstore altloc)))
                            (if (samep altvalue altstore-altvalue-not-found)
                              altvalue
                              (tagcase altvalue
                                 (Cont (M.altstore altloc))
                                 (t altvalue) ) ) ) ) ) ) )))
      (call :meaning (meaning form1)
            :cont (wrapper M.cont)
            :altstore
             (altstore-lambda (altloc)
               (let ((altvalue (M.altstore altloc)))
                 (if (samep altvalue altstore-altvalue-not-found)
```

```
altvalue
(tagcase altvalue
   (Cont (AltValue<-Cont (wrapper (Cont<-AltValue altvalue))))
   (t altvalue) ) ) ) ) ) ) ) )
```

`unwind-protect` evaluates its first parameter and ensures that the following parameters known as the cleanup forms will always be evaluated afterward even if the evaluation of the first parameter is escaped from prematurely. Like `prog1` `unwind-protect` returns what returns its first parameter. Multiple values are then transmitted if the outer continuation expects them.

When an escaping form is evaluated, cleanup forms are unrolled and evaluated one after the other until the enclosing binding form is reached or another escaping form from within cleanup forms is encountered.

```
(block foo
  (unwind-protect (return-from foo 'a)
     (return-from foo 'b) ) )
→ b
```

14 Exceptions

Exceptions are raised by the evaluator when abnormal or exceptional events are encountered. They may also be raised by the program itself and benefit of the same mechanism.

An exception involves two things: a resumption continuation (which, if it exists, is the continuation of the form where the exception was raised) and some information which describe in a more detailed way the context of the exception. Nearly all system exceptions that may be raised by the evaluator do not come with such additional information such as variable names, access to the lexical environment ...which pertain to the production environment. To provide some additional informations may although be useful for the user when extending the exception system.

Exceptions belong to the predefined extensible classe named `exception`. The class of an exception indicates its nature (`divide-by-zero`, `unfound-identifier` ...). The evaluator does not always create fresh instances of exception classes when raising a system exception. The exception handling mechanism is open: the program may have its own classes of exception which must inherit from the `exception` class, and it may raise instances of these.

Two kinds of exceptions exist:

continuable exception When a continuable exception is raised, it is possible to continue the suspended computation i.e. send value(s) to the continuation representing the suspended computation.

non continuable exception When a non continuable exception is raised, the evaluator is unable to accompany it with a resumption continuation. It is therefore not possible to resume such a continuation. The handler must then mandatorily escape by `throw`, `go` or `return-from`.

Two functions are provided to raise continuable (`cerror`) or non continuable exceptions (`error`). An exception is known to be continuable if it answers true under the `continuable-p` predicate. If the exception is continuable then the continuation of the handler is the resumption of the exception, the value(s) that may produce the handler are then returned where the exception was raised. Therefore the continuability of an exception is not a property of its meaning but a result of the implementation and its ability to determine the right continuation. As a matter of fact some predefined exceptions such as `division-by-zero` may be continuable on a computer and not continuable on another. That depends on the properties of the floating point unit.

Resumption continuations always have a dynamic extent and thus can only be invoked during the evaluation of the exception handler.

Forms are always evaluated under a given *exception handler* which is applied whenever an exception is raised. Its behaviour is the program answer to the exception. An initial exception handler exists which only task is to abort the whole computation. This initial handler catches all exceptions. It is good programming style to shadow this drastic behaviour with its own exception handler. The scope of an exception handler is dynamic, unless shadowed by another exception handler, it handles all exceptions raised during the evaluation of its body. Exception handlers are not closed under abstraction creation.

The handling of an exception is thus the result of an interaction between the evaluator raising an exception which may or may not be continued (and only the evaluator knows if it possible depending on its internal state) and the current exception handler which, knowing the context of the occurrence of the exception, can decide wether to escape to a safer level or to retry the computation after some possible clean-up.

14.1 Exception Related Domains

14.2 Exception Features

Special Form **`with-handler`**

(`with-handler` ((*type-name function*)
...)
forms...)

```
(def (meaning (with-handler-form type-names functions body))
  (build :meaning M
    (let ((types (types<-names type-names M.typenv)))
      (if (samep types types-not-found)
        (exit :message "Type not found")
        (call :meaning (meaning functions)
              :cont (build :cont C
                      (call :meaning (meaning body)
                            :handlers (*insert (compose-handlers types C.values)
                                               M.handlers )
                            :cont M.cont ) ) ) ) ) ) )
(define (compose-handlers types funcs)
  (if (*null types)
    no-handler
    (extend (compose-handlers (*tail types) (*tail funcs))
            (*head types)
            (*head funcs) ) ) )

(define (lookup-handler type handlerenv)
  (let ((func ((*head handlerenv) type)))
     (if (samep func no-handler)
       (lookup-handler type (*rest handlerenv))
       func ) ) )
```

The first parameter of the `with-handler` form is a sequence of clauses composed of a *type-name* and an associated function. If the first term of the clause is not a subtype of `exception` then the non-continuable exception `incorrect-type` will be raised. If the second term is not a function which can accept one argument then the non-continuable `not-a-function` or `incorrect-arity` exception will be raised. All these clauses are considered to form the current handler.

The following forms are considered as an implicit `progn` and forms the body of `with-handler`. If no exception is raised during the evaluation of the body, the `with-handler` form returns whatever returns its body, `with-handler` can thus transmit multiple values. If an exception is raised then the following process will be performed: the type of the exception will be compared to the *type-name* of the first clause of the current handler. If the type of the exception is an subtype of *type-name* then the associated function will be applied with the exception as argument. Otherwise the next clause will be considered. If no clause of the current handler can handle the exception then the clauses defining the immediately enclosing handler will be considered.

The resumption continuation will be the continuation of the handler invocation. The exception handler allows escape from the erroneous context with any known escaping form, it can also resume the exception, provided it is continuable, and submit some values(s) to the resumption continuation. The application of the current handler is done in the dynamic context of the exception and under the protection of the handler which was current when the last `with-handler` form installed the current handler. With implementation words, the current handler is evaluated on top of the stack without any premature escape. That property allows handlers to decline handling an exception by raising it again. The next enclosing exception handler is then applied on this same exception.

```
(with-handler ((Integer (lambda (e) t)))
  (with-handler ((Integer (lambda (e)
                          (not (if (continuable-p e)
                                   (cerror e)
                                   (error e) )) )))
    (cerror 813) ) ) → ()
(with-handler ((Integer (lambda (e) t)))
  (with-handler ((Symbol (lambda (e)
                          (not (if (continuable-p e)
                                   (cerror e)
                                   (error e) )) )))
    (cerror 813) ) ) → t
```

If the exception is not continuable, trying to resume it raises the non-continuable `non-continuable-exception` exception.

Function **cerror**

(cerror *exception*)

```
(def (meaning (cerror-form form))
  (build :meaning M
    (call :meaning (meaning form)
          :cont (build :cont C
                  (if (= (*length C1.values) 0)
                     (exit :message "Empty multiple values")
                     (let ((fn (lookup-handler (type<-value (first C.values))
                                               M.handlerenv )))
                       (call :func fn
                             :values (Values<-Value (first C.values))
                             :cont M.cont
                             :handlers (*rest M.handlers) ) ) ) ) ) ) )
```

`cerror` raises the exception which is the value of its first argument. The resumption continuation of this exception is the current continuation of the `cerror` form. The current continuation will be resumed if the exception is continued. The resumption continuation has a dynamic extent i.e. can only be resumed during the application of the handler.

When the exception is raised, the current handler is applied on it, the second argument being a true value since `cerror` raised continuable exceptions.

Function **error**

(error *exception*)

```
(def (meaning (error-form form))
  (build :meaning M
    (call :meaning (meaning form)
          :cont (build :cont C
                  (if (= (*length C1.values) 0)
                     (exit :message "Empty multiple values")
```

```
(let ((fn (lookup-handler (type<-value (first C.values))
                          M.handlerenv )))
  (call :func fn
        :values (Values<-Value (first C.values))
        :cont (build :cont C2
                     (exit :message "Non continuable exception") )
        :handlers (*rest M.handlers) ) ) ) ) ) ) )
```

`error` raises the exception which is the value of its first argument. The raised exception is not continuable. To try to resume a non continuable exception signals the non continuable `non-continuable-exception` exception.

When the exception is raised, the current handler is applied on it, the second argument being a false value since `error` raised non continuable exceptions.

Predicate `continuable-p`

(`continuable-p` *exception*)

This function tests if an exception is continuable or not. This function returns a boolean.

14.3 Interruptions

Asynchroneous interruptions are handled by the exception model. When an interruption arrives (a keyboard character, a mouse click, a clock tick ...) an exception is created (or not, depending on the fact that the implementation creates freshes exceptions) and shortly after that very exception is raised. The way to raise it is to choose a form π which meaning is $[\![\pi]\!]$ and to logically replace it by $[\![$ `(progn (cerror` *exception*`)` π`)` $]\!]$.

14.4 Predefined Exceptions

All predefined exceptions are listed hereafter. For each of them is indicated its name, the circumstances of its occurrence and if it is continuable. All these exceptions are instances of the `exception` class. As can be noted no elaborate information accompany them. The reason is that we are in a run-time Lisp and most information is only useful for debuggers in production mode and would cost much if recorded in the execution image.

`clock`	continuable	raised every tick by the evaluator
`user-interrupt`	continuable	raised asynchroneously by the user
`non-continuable`	continuable	raised on a trial to resume a non continuable exception
`undefined-function`	non resumable	raised if a function is imported from an unloaded module.
`module-redefinition`	continuable	raised if one tries to redefine a module.
`illegal-import`	non resumable	raised if ones tries to import from a module something which is not defined in this module.
`illegal-export`	non resumable	raised if one tries to export something which is not defined in the current module.
`undefined-module`	continuable	raised if one tries to load a module which does not exist.
`storage-overflow`	not continuable	raised if no more memory
...	...	...

15 Threads

Threads allow multiple evaluations to be run concurrently in a coroutine manner. No provision is made for multi-processors.

Threads are created in the lexical scope of a scheduler. Threads are first-class objects but their interesting use is restricted to the dynamic extent of their scheduler, a premature exit out of this scheduler implicitely kills the whole set of threads managed by this very scheduler: they cannot be resumed anymore. Threads may share a common part of lexical or dynamic context thus providing privately shared controlled resources.

15.1 Thread Related Domains

```
(def-domain "Thread" (* Cont
                        (label monitor AltLocation) ))

(def-domain "Monitor" (* (label scheduler Func)
                         (label status (enum alive-monitor dead-monitor)) ) )

(def-domain "SchEnv"
  (-> Id (lifted AltLocation schenv-AltLocation-not-found)) )
```

15.2 Thread Features

Predicate **thread-p**

(thread-p *form*)

The predicate `thread-p` returns true only if the value of its argument is a thread.

Predicate **alive-thread-p**

(alive-thread-p *form*)

```
(def-initial-function alive-thread-p
  (build :Func F
    (if (= (*length F.values) 1)
      (tagcase (first F.values)
        (Thread
         (if (samep (Monitor-Status (F.altstore (Thread-Monitor (Thread<-Value (first F.values)))))
                    monitor-alive )
           (call :cont F.cont
                 :values (Values<-Value (Value<-boolean boolean-true)) )
           (call :cont F.cont
                 :values (Values<-Value (Value<-boolean boolean-false)) ) ) )
        (t (call :cont F.cont
                 :values (Values<-Value (Value<-Boolean boolean-false)) )) )
      (exit :message "Erroneous Number of Arguments") ) ) )
```

The predicate `active-thread-p` returns true only if the value of its argument is an active thread. It is an error not to submit a thread as argument to `alive-thread-p`.

Special Form **schedule**

(schedule *schedule-label*
          monitor
          forms...)

```
(def-meaning (scheduling label form body)
  (build :meaning M
    (call :meaning (meaning form)
          :cont (build :cont C1
                   (if (= (*length C1.values) 0)
                     (exit :message "Empty multiple values")
                     (tagcase (first C1.values)
                        (Func (let* ((altloc (new-altlocation C1.altstore))
                                     (wrapper (lambda (k)
                                                (build :cont C2
                                                   (call :cont k
                                                         :altstore
                                                          (extend C2.altstore altloc
                                                             (make-monitor
                                                              :scheduler (first C1.values)
                                                              :status dead-monitor ) ) ) ) )) )
             (call :meaning (meaning body)
                   :schenv (extend M.schenv label altloc)
                   :altstore (extend
                               (altstore-lambda (altloc)
                                 (let ((altvalue (C1.altstore altloc)))
                                   (if (samep altvalue
                                              altstore-altvalue-not-found)
                                       altvalue
                                       (tagcase altvalue
                                         (Cont (AltValue<-Cont
                                                 (wrapper
                                                  (Cont<-AltValue altvalue))))
                                         (t altvalue) ) ) ) )
                               altloc
                               (AltValue<-Monitor
                                 (make-monitor
                                   :scheduler (Func<-Value (first C1.values))
                                   :status alive-monitor ) ) )
                   :cont (wrapper M.cont) ) ))
           (t (exit :message "Not a function")) ) ) ) ) ) )
```

The second parameter is evaluated and must yield a function which is able to accept at least one argument otherwise the non-continuable `not-a-function` exception will be raised. The *forms* are sequentially evaluated and the value(s) of the last one becomes the value(s) of the `schedule` form. If during that evaluation, a thread is created (see `suspend`), the computation will be suspended, the function *monitor* will be applied on the new thread and its return value(s) will become the value(s) of the whole `schedule` form. The *monitor* is run in the dynamic context of the `schedule` form. There exist an ultimate monitor which is only able to terminate the program. When the `schedule` form returns some value(s) or is escaped, all inner threads will become non resumable. The creation of a thread is not an escaping form and it is therefore allowed to suspend a whole set of threads.

Special Form `suspend`

(`suspend` *schedule-label form*)

```
(def-meaning (suspension label arguments)
  (build :meaning M
    (let ((altloc (M.schenv label)))
      (if (samep altloc schenv-altlocation-not-found)
```

```
    (exit :message "No scheduler")
    (call :meaning (meaning arguments)
          :cont (build :cont C
                   (let ((monitor (Monitor<-AltValue (C.altstore altloc))))
                     (if (samep (Monitor-status monitor) alive-monitor)
                       (call :func (Monitor-Scheduler monitor)
                             :values (*insert (Value<-Thread (make-thread
                                                              :cont M.cont
                                                              :monitor altloc ))
                                              C.values ) )
                       (wrong :message "Obsolete monitor"
                              :values (no-values) ) ) ) ) ) ) ) ) )
```

The scheduler named *schedule-label* is looked for, if it does not exist the continuable `scheduler-unfound` is raised; if it exists and is obsolete the continuable `obsolete-scheduler` exception is raised. Otherwise the second parameter is evaluated, the current computation is supended and a new thread is created. When resumed this thread will continue its computation where it was ended. The monitor associated with the *schedule-label* will be applied on this new thread and on the value(s) returned by the second parameter. It is an error if the monitor cannot accept all these arguments.

Special Form **resume**

(`resume` *thread form*)

```
(def-meaning (resumption form1 form2)
  (build :meaning M
     (call :meaning (meaning form1)
           :cont (build :cont C1
                    (if (= (*length C1.values) 0)
                      (exit :message "Empty multiple values")
                      (tagcase (first C1.values)
                         (Thread
                          (call :meaning (meaning form2)
                                :cont (build :cont C2
                                         (let ((th (Thread<-Value (first C1.values))))
                                           (if (samep (Monitor-Status
                                                        (Monitor<-Altvalue
                                                         (C2.altstore
                                                          (Thread-Monitor th))))
                                                       alive-monitor )
                                             (call :cont (Thread-Cont th)
                                                   :values C2.values )
                                             (wrong :message "Obsolete thread"
                                                    :values
                                                     (Values<-Value
                                                      (first C1.values)) ) ) ) ) ) )
                         (t (exit :message "Not a thread")) ) ) ) ) ) )
```

If the first argument is a thread then it is resumed with the value(s) returned by the second parameter given as value(s) of the corresponding `suspend` form. It is thus possible to resume a thread and to give it some informations with the desired protocol. It is an error not to provide a thread as first parameter. if the thread is obsolete then the continuable `obsolete-thread` exception will be raised.

15.3 An Example of Threads

Threads allow to implement the full generality of Scheme `call`/cc.

```
(defmodule call/cc
   (import)
   (export call/cc       ;;function
           with-call/cc  ;;macro
           )
   (defun call/cc (fn)   ;;call/cc is first class
      (funcall call/cc-invoker fn) )
   (variable-define call/cc-invoker
      (lambda (v) v) )
   (defun with-call/cc (call)
     (let ((g (gensym)))
       `(schedule ,g
           (lambda (thread cont-handler)
               (funcall cont-handler
                        (lambda (value) (resume thread value)) ) )
           (setq call/cc-invoker
                 (lambda (cont-handler) (suspend ,g cont-handler)) )
           . ,(cdr call) ) ) ) ) )
```

To use `call/cc`, a module needs only to import the `call/cc` module and to wrap the entry point in the `with-call/cc` macro. One may also wrap other forms and have partial continuations.

16 Miscellaneous Libraries

Not all libraries are listed hereafter, only a few functions are given.

no **`eval`**

There is no `eval` function in Reasonable Lisp. This is not an omission, it is an intended feature, but `apply` exists.

Function **`apply`**

`(apply` *function form... last-form*`)`

applies the first argument on the other arguments. The last argument is a list that contains all the last arguments for *function*.

```
(apply (function cons) 1 (list 2)) → (1 . 2)
(apply (function cons) (list 1 2)) → (1 . 2)
(apply (function cons) 1 2 '())    → (1 . 2)
```

Function **`read`**

`(read` *grammar*`)`

read an expression according to the grammar given as parameter.

Function **`end`**

`(end)`

```
(defsem meaning-of-end
  (build :function
      final-answer ) )
```

terminates the program.

References

[Gabriel & Pitman 88] Richard P. Gabriel, Kent M. Pitman, *Technical Issues of Separation in Function Cells and Value Cells*, Lisp and Symbolic Computation, Vol 1, Number 1, June 1988, pp 81–101.

[Gordon 75] Michael J. C. Gordon, *Operational Reasoning and Denotational Semantics*, Actes du Colloque IRIA "Constructions et Justifications de Programmes", G.Huet & G.Kahn (eds.), Arc et Senans Juillet 1975, pp 83–98.

[Muchnick & Pleban 80] Steven S. Muchnick, Uwe F. Pleban, *A Semantic comparison of Lisp and Scheme*, Lisp Conference 1980, pp 56–64.

[Rees & Clinger 86] Jonathan A. Rees, William Clinger, *Revised^3 Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 21, 12, Dec 86, pp 37 – 79.

Index