Appendix

The ABCL/1 User's Guide

Etsuya Shibayama
Yuuji Ichisugi
Akinori Yonezawa

DO NOT COPY

This User's Guide has three parts. Sections 1–7 are the ABCL/1 language manual, in which we describe the syntax and semantics of the language using program examples. Sections 8–15 are the introduction to the ABCL/1 system, in which we explain how to interact with the ABCL/1 system through session examples. Sections 16–21 are the miscellaneous part including the command dictionary of the ABCL/1 system and the formal syntax description of the language ABCL/1.

The current ABCL/1 system/language depends on Common Lisp[**?**]. For instance, the ABCL/1 system is implemented in Common Lisp and the ABCL/1 language includes most Common Lisp functions, macros, and special forms. The reader of this user's guide is expected to have some basic knowledge of Common Lisp such as its syntax and basic functions.

Contents

PART I **4**

1 The Computation Model 4

2 Object Definitions 6

2.1 Message Patterns 8
2.2 Reply Destination Variables 10
2.3 Sender Variables 11

2.4 Constraints . . . 11
2.5 Behavior Descriptions . . . 11

3 Basic Forms . . . 12
3.1 Assignment Forms . . . 12
3.2 Object Definition Forms . . . 12
3.3 Reply Forms . . . 13
3.4 Common Lisp Forms . . . 13
3.5 Bracket Forms . . . 15

4 Message Passing Forms . . . 16
4.1 Message Passing in the Ordinary Mode . . . 16
4.2 Past Type Message Passing Including a Reply Destination . . . 18
4.3 Future Objects . . . 20
4.4 Multicast . . . 23
4.5 Message Passing in the Express Mode . . . 23
4.6 Atomic Forms . . . 24
4.7 Non-Resume Forms . . . 26
4.8 Parallel Message Passing Forms . . . 26

5 Control Structures . . . 28
5.1 Common Lisp Control Structures . . . 28
5.2 Wait-For Forms . . . 29
5.3 Wait-For-Loop Forms . . . 30
5.4 Match Forms . . . 32
5.5 Match-Loop Forms . . . 33
5.6 Routine Calls . . . 33
5.7 Defining Macros . . . 35

6 Miscellaneous Forms . . . 38
6.1 Suicide . . . 38
6.2 Break Points . . . 38

7 Variables and Their Scope Rule . . . 39
7.1 Variables in the Global Environment . . . 39
7.2 State Variables . . . 40
7.3 Temporary Variables . . . 41
7.4 Pattern Variables . . . 42
7.5 Environment Variables . . . 42
7.6 Pseudo Variables . . . 43

PART II **44**

8 The ABCL/1 System 44
8.1 Invocation of the ABCL/1 System 44
8.2 Halting the ABCL/1 System 45

9 The Top Level 45
9.1 The Help Command 45
9.2 Object Definition at the Top Level 45
9.3 Evaluation of Forms at the Top Level 46

10 The Loader and Compiler 47
10.1 Loading an ABCL/1 Program 47
10.2 Compiling an ABCL/1 Program 48

11 Getting Information about Objects 49
11.1 Listing the Objects Defined at the Top Level 49
11.2 Describing Objects 49
11.3 Listing the Message Protocols of an Object 50

12 Resetting Objects 50

13 The Inspector 51
13.1 The Help Command in the Inspector 51
13.2 Listing Information about the Inspected Object 52
13.3 Halting the Inspector 53
13.4 Interruption from the Keyboard 54
13.5 Recursive Invocation of the Inspector 55
13.6 Listing the Active Objects 56
13.7 Continuing the Suspended Computation 57
13.8 Getting the Contents of the Program Counters of Objects . . 57

14 Execution Monitoring 59
14.1 The Tracer 59
14.2 Recording Event Histories 61
14.3 Break Points 62
14.4 Stepwise Execution 64

15 Other Top Level Forms 66

16 The Summary of the Top Level Forms 67

17 The Summary of the Inspector Commands 69

PART III **71**

18 Syntax of ABCL/1 71

19 The ABCL/1 Mode in GNU Emacs 74

20 Known Bugs and Features 75

21 Caution!!! 76

PART I

1 The Computation Model

Objects in ABCL/1 are autonomous information processing agents which work cooperatively with one another. Each object has its own computing power and works concurrently (or in parallel) with other objects. The computation model on which ABCL/1 is based assumes neither shared memory nor global clock. A collection of objects constitute a distributed computing system.

An object has its own internal world, which consists of a local persistent memory and procedures inquiring/updating the local memory. They are called *state* and *script*, respectively. Also we assume that an object has its own local clock, namely, the local time is a well-defined concept for the object. The internal world of an object is a protected structure and cannot be accessed directly from any other object.

Objects interact with one another via *message passing*. In response to a received message, an object executes one of the procedures in its script. Execution of a procedure by an object is a sequence of the following actions:

1. inquiring and updating the *state* of the object,
2. creating new objects,
3. sending and accepting messages, and
4. returning a value as a reply to a received message.

In ABCL/1, the third kind of actions subsumes the fourth.

For each object, there is a unique *message queue* and arriving messages are put in the message queue. We call this event *message arrival*. Messages in a message queue will be processed one at a time in a sequential manner. In ABCL/1, message arrival is asynchronous with the actions 1, 2, 3, and 4 mentioned above. This means that messages can even arrive at an object in deadlock.

Each object is in one of the three modes, *dormant*, *waiting*, and *active* at any time. An object is in the *dormant* mode at its *birth* time. It becomes *active* when receiving a message. During execution of a procedure, it is in the *active* mode. It enters the *waiting* mode when it needs to receive another message in order to proceed with the execution of the procedure. By receiving an expected message, the object becomes *active* again, and continues the computation. After completing the procedure, the object becomes *dormant* again.

There are three types of message passing, *past*, *now*, and *future*[1]. Just after transmitting a *past* type message, the sender object, say O, can continue its computation. If the receiver object, say O' is in the *dormant* mode, O' receives and then processes the message concurrently with the execution of O. In this case, O does not expect any reply as a response to the message.

In contrast with the past type message passing, the sender object of a now type message does not resume its computation until the reply to this message arrives. In this case, the receiver object may return the reply while processing the message as well as after completing the actions in response to this message (see in [?] for detail).

The sender object O of a future type message also expects replies. But O does not have to wait for the replies immediately after the message transmission. O can continue its current sequence of actions. Each reply will be received asynchronously with these actions. For the purpose of asynchronous message reception, O creates a special object called a *future object* f whose behavior is similar to a queue. O attaches the name (or destination) of f to a future type message. All the replies to the message will arrive at f. O can remove and get replies stored in f and also check whether f is empty or not, whereas any other object O' cannot read the contents of f. What O' can do best is sending back a reply to f.

In ABCL/1, a reply is returned as a message. For this convention, each message which requires a reply contains the information about the place to which the reply will be sent back. We call this information a *reply destination*.

Each message is sent in either the *ordinary* or *express* mode. A message sent in the express mode has higher priority than those sent in the ordinary mode. More precisely, if an object receives a message in the express mode while processing a message in the ordinary mode, the object suspends the current computation sequence and starts processing the express message. By default, the suspended computation will be resumed after completing the actions for the express message.

ABCL/1 satisfies the *transmission ordering law*:

> Suppose that two messages M and M' have the same sender O and the same receiver O'; If M and M' are sent in this order according to the local

[1] In the rest of this user's guide, we use "past (now, future) type message." This means that the message is sent as past (now, future) type message passing. It is *not* implied that each *message* has a type.

clock of O, M and M' are always received in the same order according to the local clock of O'.

Note that, in general, if M and M' are sent from different objects and received by the same one, their arrival order cannot be determined.

2 Object Definitions

In the language ABCL/1, an object whose name is *object-name* is defined in the following form:

```
[object object-name
  (state [state-variable := initial-value] ···)
  (script
    (=> message-pattern @ reply-destination-variable
                        from sender-variable
                        where constraint
        (temporary [temporary-variable := initial-value] ···)
        behavior-description)
        ⋮
    (=>> message-pattern @ reply-destination-variable
                        from sender-variable
                        where constraint
        (temporary [temporary-variable := initial-value] ···)
        behavior-description)
        ⋮
        )
  (routine
    (routine-name argument-list
         behavior-description)
        ⋮
    (routine-name argument-list
         behavior-description))]
```

In the syntax descriptions of this user's guide, a terminal symbol is written in a `type` face and a non-terminal symbol in an *italic* face.

- The *state-variable*s of an object are the variables which represent the internal persistent *state* of the object.
- An object defined in the above form accepts a message which matches some *message-pattern* and satisfies the corresponding *constraint*. When an ordinary

(or express) mode message arrives, message patterns and constraints following `=>` (or `=>>`, respectively) are examined in the *top-to-bottom* order. At that time, *reply-destination-variable* and *sender-variable* are bound to the reply destination and the sender of the message, respectively. A *constraint* may contain the *reply-destination-variable* and *sender-variable*.

- After accepting a message, the object performs a sequence of actions described in the corresponding *behavior-description* part. While performing these actions, the object can use the corresponding *temporary-variable*s.
- If some state (or temporary) variable need not be initialized, its declaration [*variable-name* `:=` *initial-value*] can be replaced as *variable-name*. In such a case, its initial value will be `nil`.
- A *routine* is a local function[2] which can be used only within the object.
- The state and temporary variable declaration parts, the reply destination variables, sender variables, the constraints attached to the message patterns, and the routine declaration part are all optional.

The following example defines a tiny object which accepts a `[:hello]` message in the ordinary mode and returns the value `:hi`.

```
[object greeting
  (script
   (=> [:hello]
       !:hi))]
```

In ABCL/1, according to the keyword convention of Common Lisp, each symbol whose name begins with a colon (`:`) is treated as a constant symbol such that its value is always its name. In the behavior description part of this object, we use the `!`-notation, which means that this object returns `:hi` as a reply to a `[:hello]` message.

The above definition does not contain any state variable declaration part. Also the message pattern has no constraint. The object `greeting` has no internal state and its behavior is purely functional in the sense that it always behaves in the same way in response to the same messages.

The next definition contains the state variable declaration part.

```
[object counter
  (state [c := 0])
  (script
```

[2]This does not mean a mathematical function but a function in the sense of Common Lisp.

```
(=> [:increment]
    [c := (1+ c)])
(=> [:value]
    !c)
(=> [:reset]
    [c := 0]))]
```

The object `counter` has a single state variable `c` whose initial value is `0`. The state variable `c` is considered to represent the contents of the counter. This object accepts three kinds of messages:

- `[:increment]` for increasing the contents of the counter by one,
- `[:value]` for retrieving the contents of the counter, and
- `[:reset]` for setting the contents of the counter to zero.

The forms `[c := (1+ c)]` and `[c := 0]` are assignment forms.

2.1 Message Patterns

A *message pattern* consists of constant symbols and pattern variables. Any symbol in a pattern with a leading colon (`:`) is a constant. Also the numbers and the symbols `t` and `nil` are regarded as constants. Each other symbol in a message pattern is a pattern variable. A pattern variable matches any value, whereas a constant matches only itself.

ABCL/1 has a *pattern constructor*. Suppose that $pattern_1$, $pattern_2$, $\cdots$, and $pattern_n$ are valid message patterns. The following is also a valid message pattern:

$$[pattern_1\ pattern_2\ \ldots pattern_n]$$

This message pattern matches a message[3]:

$$[message\text{-}component_1\ message\text{-}component_2\ \cdots\ message\text{-}component_n]$$

[3]Exactly speaking, a message pattern:

$$[pattern_1\ pattern_2\ \ldots pattern_n]$$

and the *value* of:

$$[message\text{-}component_1\ message\text{-}component_2\ \cdots\ message\text{-}component_n]$$

match under the condition mentioned above. This value in turn is equivalent to the value of:

$$\texttt{(list}\ message\text{-}component_1\ message\text{-}component_2\ \cdots\ message\text{-}component_n\texttt{)}$$

(see Section 3.5).

such that the i-th pattern $pattern_i$ and the i-th $message\text{-}component_i$ match ($1 \leq \forall i \leq n$). For instance, a message pattern `[:add n]` matches a message `[:add 4]` and the pattern variable `n` is bound to `4` after the match.

The following example is the definition of a new counter object. On reception of a message of length 2 whose first component is `:add`, this object increments its contents by the value specified as the second component of the message. For instance, if the object receives a message `[:add 2]`, it will increment its contents by `2`.

```
[object counter
  (state [c := 0])
  (script
   (=> [:add n]
       [c := (+ c n)])
   (=> [:value]
       !c)
   (=> [:reset]
       [c := 0]))]
```

In general, a pattern variable is treated as a *read only* variable in the corresponding behavior description part.

ABCL/1 supports another pattern constructor. Suppose again that $pattern_1$, $pattern_2$, $\cdots$, and $pattern_n$ are valid message patterns. Then the following is also a valid message pattern:

$$[pattern_1 \cdots pattern_{n-1} \ . \ pattern_n]$$

This message pattern matches a message:

$$[message\text{-}component_1 \cdots message\text{-}component_m]$$

such that:

1. $n - 1 \leq m$,
2. the $pattern_i$ matches the $message\text{-}component_i$ ($1 \leq \forall i \leq n - 1$) and
3. $pattern_n$ matches $[message\text{-}component_n \cdots message\text{-}component_m]$.

For instance, a message pattern `[first second . rest]` matches a message `[:a :b :c :d]` and, after this pattern matching, the pattern variables `first`, `second`, and `rest` are bound to `:a`, `:b`, and `[:c :d]`, respectively.

In order to deal with optional components of messages, the last pattern constructor of ABCL/1 is introduced. The message pattern:

$$[keyword\ variable_1 \cdots variable_n\ \texttt{\&}\ variable_{n+1} \cdots variable_{n+n'}]$$

where *keyword* is a symbol with a leading colon (`:`) and *variable*s are pattern variables matches a message:

$$[message\text{-}component_0 \cdots message\text{-}component_m]$$

such that:

1. $n \leq m \leq n + n'$,
2. *message-component*$_0$ is *keyword*, and
3. the *pattern-variable*$_i$ matches the *message-component*$_i$ $(1 \leq \forall i \leq m)$.

After this pattern matching, *pattern-variable*$_{m+1}$ through *pattern-variable*$_{n+n'}$ are bound to `nil`. For instance, a message pattern `[first second & third fourth]` matches a message `[:a :b :c]` and, through this pattern matching, the pattern variables `first`, `second`, `third`, and `fourth` are bound to `:a`, `:b`, `:c`, and `nil`, respectively.

2.2 Reply Destination Variables

A *reply destination variable* is a pattern variable which is bound to the reply destination of an incoming message. A reply to a message M is nothing but a past type message to the reply destination of M. The following part of the definition of the object `counter`:

```
(=> [:value]
    !c)
```

is equivalent to:

```
(=> [:value] @ reply-to
    [reply-to <= c])
```

where the form `[reply-to <= c]` means that a message `c` of past type is sent to the reply destination `reply-to`[4].

[4]More precisely, `[reply-to <= c]` means that a value assigned to `c` is sent as a past type message to the reply destination to which `reply-to` is bound.

2.3 Sender Variables

The sender of a message can be captured by the following expression:

```
(=> message-pattern from sender-variable
    ... )
```

or

```
(=>> message-pattern from sender-variable
    ... )
```

When a message is matched against the *message-pattern*, the *sender-variable* is bound to the sender of the message.

2.4 Constraints

The *constraint* following a message pattern is a Common Lisp form[5], which is evaluated under the environment including the variable bindings made by the corresponding pattern matching. According to the conventions of Common Lisp, a constraint is defined to be *satisfied* if its evaluation value is not `nil`.

The following object has a message pattern with an attached constraint. With this, a value by which the object increments its contents must be a positive integer.

```
[object counter
  (state [c := 0])
  (script
   (=> [:add n] where (and (integerp n) (plusp n))
       [c := (+ c n)])
   (=> [:value]
       !c)
   (=> [:reset]
       [c := 0]))]
```

2.5 Behavior Descriptions

A behavior description part consists of several kinds of forms such as:

- assignment forms for updating the value of state/temporary variables,
- object definition forms for dynamically creating objects.
- reply forms for returning replies,

[5]Constraints should be side-effect free.

- message passing forms for sending messages,
- wait-for(-loop) forms for entering the waiting mode and specifying the acceptable messages at the mode,
- routine calls for invoking private routines, and
- Common Lisp forms.

Control structures are expressed in terms of Common Lisp forms.

In Sections 3–6, we will describe the syntax and semantics of these forms and other miscellaneous forms which can occur in behavior description parts.

3 Basic Forms

3.1 Assignment Forms

In ABCL/1, the following *assignment* form:

[*variable* := *form*]

means that the evaluation result of the *form* is assigned to the *variable*. Notice that this form is equivalent to the Common Lisp form:

(setq *variable form*)

3.2 Object Definition Forms

An object definition form (see Section 2):

```
[object object-name
  (state ···)
  (script ···)
  (routine ···)]
```

occurs either in some other form or at the top level of a source program. In the former case, by each evaluation of this form, an object is created whose behavior is specified by the form and returned as the evaluation result. In this case, the *object-name* is a local name of the newly created object (therefore, any other object cannot refer to this name) and this part can be omitted.

In the latter case, by evaluation of this form, an object with the global name *object-name* is created. The created object can be referred to by any object (including itself) through its global name *object-name*.

3.3 Reply Forms

By a *reply* form,

!*form*

the evaluation result of the *form* is sent back as a reply to the currently processed message.

A value returned by the reply form can be an object.

```
[object create-stack
  (script
    (=> [:new]
        ![object
           (state [stack := nil])
           (script
             (=> [:push x]
                 [stack := (cons x stack)])
             (=> [:pop]
                 !(car stack)
                 [stack := (cdr stack)])
             (=> [:top]
                 !(car stack))
             (=> [:reset]
                 [stack := nil]))]))]
```

When the object `create-stack` defined above receives a `[:new]` message, it creates and returns an object whose behavior is described as the `[object ...]` form following an exclamation mark (`!`).

When the user likes to use several stack objects, it is more convenient to define the above object than to define a necessary number of stack objects independently of one another.

3.4 Common Lisp Forms

In order to describe the behavior of an object, not only ABCL/1 constructs, but also Common Lisp functions, macros, and special forms are used.

3.4.1 Function Calls

A *function call* is a Common Lisp form whose first element is a symbol which names a *function*. Upon evaluation of a function call whose first element names a function f, all the remaining elements are evaluated in the left to right order and then f is

block	function	macrolet	return-from
catch	go	multiple-value-call	setq
compiler-let	if	multiple-value-prog1	tagbody
declare	labels	progn	the
eval-when	let	progv	throw
flet	let*	quote	unwind-protect

Figure 1: The Names of the Common Lisp Special Forms

applied to the evaluation results. ABCL/1 supports all the functions (and thus, all the data types also) of Common Lisp.

In the following example:

```
[object printer
  (script
   (=> any-message
       (princ any-message)
       (terpri)))]
```

an object `printer` is defined, which receives any message and prints it using the Common Lisp functions `princ` and `terpri`. Notice that a message pattern `any-message`, which is a single pattern variable, matches any message.

The current implementation of ABCL/1 does not support any I/O and arithmetic functions by itself. In order to describe such actions of an object, we must use Common Lisp forms.

With the Common Lisp function `format`, an object with the same behavior can also be defined as follows:

```
[object printer
  (script
   (=> any-message
       (format t "~A~%" any-message)))]
```

3.4.2 Special Forms

A special form of Common Lisp is a form whose first element is one of the 24 symbols in Figure 1. Special forms are usually used as control structures (see Section 5.1). The evaluation strategy of a special form depends on its first element.

ABCL/1 currently supports just a subset of the Common Lisp special forms. Each symbol in Figure 2 can be the first element of an ABCL/1 special form.

block	if	multiple-value-prog1	setq
declare	let	progn	tagbody
function	let*	quote	
go	multiple-value-call	return-from	

Figure 2: The Names of the Special Forms Available in ABCL/1

3.4.3 Macros

Common Lisp implements a *macro* expansion mechanism. A *macro call*, a form whose first element is a *macro symbol*, is expanded into another form and then evaluated.

Most built-in macros of Common Lisp can be used in ABCL/1 programs. However, some macro forms are expanded into other forms including special forms which are not supported by the current implementation of ABCL/1. For instance, a *with-open-file* form is possibly expanded into a form including an *unwind-protect* form, whose correct execution is not guaranteed by the current implementation.

A macro is defined using a *defmacro* form of Common Lisp. More about macros is described in Section 5.7.

3.5 Bracket Forms

A bracket form:

$$[form_1 \cdots form_n]$$

in a behavior description part is just an abbreviation of the Common Lisp form[6]:

$$(\texttt{list}\ form_1 \cdots form_n)$$

Bracket forms are often used to compose messages from message components. For instance, the following message passing form:

```
[stack-object <= [:push 5]]
```

will send the value of a message `[:push 5]`, which is equivalent to the value of `(list :push 5)`, to the object `stack-object`.

ABCL/1 supports another kind of bracket form. The following one:

$$[form_1 \cdots form_{n-1}\ .\ form_n]$$

is an abbreviation of the Common Lisp form:

[6]Strictly speaking, the $form_1$ and $form_2$ must not be some of the reserved symbols. For instance, the $form_1$ must not be the symbol `object` and the $form_2$ must not be the symbol `:=`.

(cons $form_1$ (cons $\cdots$ (cons $form_{n-1}$ $form_n$)$\cdots$))
or equivalently
(list* $form_1$ $\cdots$ $form_{n-1}$ $form_n$)

Therefore, the following portion of the definition of a stack object:

```
(=> [:push x]
    [stack := (cons x stack)])
```

can be replaced as:

```
(=> [:push x]
    [stack := [x . stack]])
```

4 Message Passing Forms

4.1 Message Passing in the Ordinary Mode

By the following message passing forms, the *message* of types past, now, and future, respectively, in the ordinary mode is sent to the object *target*.

[*target* <= *message*]	(past)
[*target* <== *message*]	(now)
[*target* <= *message* $ *future-object*]	(future)

In the above forms, each *message* can be any form and the evaluation result of the *message* is the actual message to be transmitted. Each *target* is a form whose evaluation result is an object or a tree (*i.e.*, a possibly nested list) of objects. When the value of the *target* in a message passing form is a tree of objects, the message is sent to the objects in parallel. This kind of message passing is called *multicast*. If the tree is `nil`, the message is sent to *nothing* or, virtually, the *null* object which receives any message but does nothing.

The evaluation result of a past or future type message passing form is exactly the same as the value of a Common Lisp form `(values)`. In this case, the evaluation terminates just after the message is transmitted.

On the other hand, the evaluation result of a now type message passing form whose target is a single object is equivalent to the reply[7] which is sent back in response to the transmitted message. In this case, the evaluation of this form is suspended until

[7] In ABCL/1, the receiver of a now type message can send back more than one replies. In such a case, the returned replies except the first one will be discarded by the sender of the now type message. When an object receives more than one replies to a single now type message, the current ABCL/1 system prints a warning.

the reply arrives. Suppose that `fact` is an object which receives a message in the form of `[:fact` *number*`]` and returns the factorial of *number* as the reply. The evaluation result of the now type message passing form:

```
[fact <== [:fact 4]]
```

is `24`.

When the target of a now type message is a tree of objects, its evaluation result is the tree of the same shape such that each leaf is the reply to the message transmitted to the corresponding leaf of the target[8]. The evaluation of this form is suspended until all the replies arrive. The evaluation result of the message passing form:

```
[[fact [fact]] <== [:fact 4]]
```

is (the value of) `[24 [24]]`.

In case of future type message passing, the returned replies are put into the *future-object*. The details of future type message passing and future objects will be described in Section 4.3

The following object definition contains several past and now type message passing forms. The object `stack-machine-interpreter` interprets a message stream which represents an arithmetic expression in the reverse polish notation. We assume that the object `create-stack` creates and returns a stack object in response to a `[:new]` message.

```
[object stack-machine-interpreter
  (state [s := [create-stack <== [:new]]])
  (script
   (=> [:start]
       [s <= [:reset]])
   (=> [:add]
       (temporary
        [arg2 := [s <== [:pop]]]
        [arg1 := [s <== [:pop]]])
       [s <= [:push (+ arg1 arg2)]])
   (=> [:difference]
       (temporary
        [arg2 := [s <== [:pop]]]
        [arg1 := [s <== [:pop]]])
       [s <= [:push (- arg1 arg2)]])
   (=> number where (numberp number)
       [s <= [:push number]])
```

[8] If the corresponding leaf of the target tree is `nil`, no message transmission occurs. In this case, we consider that "the reply" is `nil`.

```
(=> [:end]
    ![s <== [:top]]))]
```

For instance, by receiving the messages:

`[:start]`, `10`, `3`, `4`, `[:add]`, `[:difference]`, and `[:end]`

the object `stack-machine-interpreter` returns the value `3`, which is the evaluation result of the arithmetic expression: $10 - (3 + 4)$.

Generally, each temporary variable is initialized every time just before the corresponding behavior description part is executed.

4.2 Past Type Message Passing Including a Reply Destination

Past type message passing may include a reply destination[9].

By the following form,

[*target* `<=` *message* `@` *reply-destination*]

the *message* of the past type with the *reply-destination* is transmitted to the *target*. The *reply-destination* is a form whose value is either an object or a tree of objects.

By using past type message passing with a reply destination, the following portion:

```
(=> [:end]
    ![s <== :top])
```

in the definition of the object `stack-machine-interpreter` can be improved as follows:

```
(=> [:end] @ reply-to
    [s <= [:top] @ reply-to])
```

In the former case, the computation proceeds as follows (Figure 3):

1. `some-object` sends an `[:end]` message to `stack-machine-interpreter`,
2. `stack-machine-interpreter` sends a now type message `[:top]` to the `stack` object to which the variable `s` is assigned.
3. the `stack` object returns its top element to `stack-machine-interpreter`,
4. `stack-machine-interpreter` receives the top element, and then
5. returns it to the reply destination of the `[:end]` message,

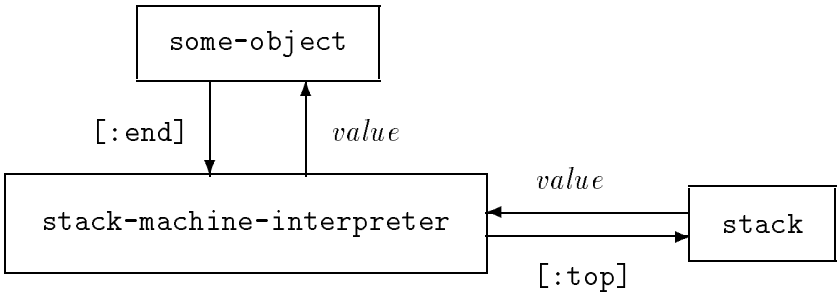

Figure 3: Prior to Introduction of the Reply Destination Mechanism

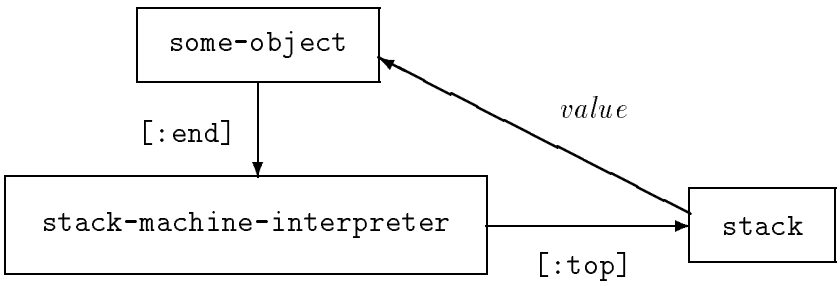

Figure 4: Posterior to Introduction of the Reply Destination Mechanism

whereas, in the latter case (Figure 4):

1. `some-object` sends an `[:end]` message to `stack-machine-interpreter`,
2. `stack-machine-interpreter` sends to the `stack` object (to which the variable `s` is assigned) a past type message `[:top]` with the reply destination attached to the `[:end]` message (*i.e.*, the value of `reply-to`)
3. the `stack` object returns its top element to the reply destination of the `[:top]` message, which also is the reply destination of the `[:end]` message.

Obviously, the total number of message passing (including returning replies) in the latter case is less than that of the former case. Furthermore, in the latter case, just after sending a `[:top]` message to the `stack` object, `stack-machine-interpreter` can accept the next message.

4.3 Future Objects

A *future object* is a special object, which is specified as the reply destination of future type message passing. Each future object f is created by another object, called the *owner* of f, using a *make-future* form:

`(make-future)`

or a *reset-future* form:

`(reset-future` *variable*`)`

By the former form, a future object is created and returned as the evaluation result of the form. By the latter form, a future object is created and assigned to the *variable*. No one except the owner of a future object f has the privilege to specify f as the reply destination of future type message passing.

ABCL/1 supports the following functions:

- `ready?`
- `next-value`
- `all-values`

to access a future object. Any object except the owner of a future object f cannot specify f as the first argument of these forms. In the following syntax descriptions, *option* and *options* are optional argument(s).

[9] In contrast, now or future type message passing *always* includes a reply destination. Now type message passing includes an implicit reply destination. In case of future type message passing, the specified future object is the reply destination.

`(ready?` *future-object*`)`
: If at least one reply is stored in the *future-object*, the value of this form is `t`. Otherwise, `nil`.

`(next-value` *future-object option*`)`
: The value of this form is the first element stored in the *future-object*. When the *future-object* is empty, the owner object waits until a reply arrives. With the `:remove t` option[10], the returned element is removed from the *future-object*, whereas, with the `:remove nil` option, it still remains even after the evaluation of this form. The default option is `:remove t`.

`(all-values` *future-object options*`)`
: The value of this form is the list of all the elements currently stored in the *future-object*. When the *future-object* is empty, this form returns `nil` if the `:wait nil` option is given. With the `:wait t` option, the object waits until at least one reply arrives. The default option of them is `:wait t`. With the `:remove t` option, all the elements are removed. The options `:remove t` and `:remove nil` have the similar meaning as those of a `next-value` form.

In the following object definitions, future type message passing, `make-future` and `next-value` forms are used. The object `merger` receives a message including two objects which are created by the object `create-sorted-object` and represent two *sorted-list*s. In response to this message, `merger` sends a future type message `[:get-all]` to each sorted-list object in order to inquire all of its contents. On reception of the `[:get-all]` message, a sorted-list object sends back its elements one by one in the increasing order to the object `merger` and also sends back a `:finished` following these elements. Then the object `merger` merges two streams stored in its future objects and sends back all the elements in the streams one by one in the increasing order to the reply destination of `merger`.

Notice that, in the following example, Common Lisp control structures `block`[11], `loop`[12], `cond`[13], `return-from`[14], and `dolist`[15] are used (see Section 3.4).

```
[object merger
  (script
   (=> [:merge sorted-list1 sorted-list2]
       (temporary
```

[10] According to Common Lisp conventions, this kind of option should be called *keyword parameters*.
[11] A *block* special form creates a *named lexical block*.
[12] The *loop* macro is an *infinite loop* construct of Common Lisp.
[13] The *cond* macro is a conditional branch construct of Common Lisp.
[14] A *return-from* special form exits from the specified named lexical block.
[15] The *dolist* macro is a loop construct which traverses a given list and applies its body to each element of the list.

```
          [future1 := (make-future)]
          [future2 := (make-future)]
          elm1 elm2)
         [sorted-list1 <= [:get-all] $ future1]
         [sorted-list2 <= [:get-all] $ future2]
         [elm1 := (next-value future1)]
         [elm2 := (next-value future2)]
         (block merger-loop
           (loop
            (cond ((eq elm1 :finished)
                   (loop
                    (if (eq elm2 :finished)
                        (return-from merger-loop))
                    !elm2
                    [elm2 := (next-value future2)]))
                  ((eq elm2 :finished)
                   (loop
                    (if (eq elm1 :finished)
                        (return-from merger-loop))
                    !elm1
                    [elm1 := (next-value future1)]))
                  ((< elm1 elm2)
                   !elm1
                   [elm1 := (next-value future1)])
                  (t !elm2
                     [elm2 := (next-value future2)])))))
       !:finished))]

[object create-sorted-list
  (script
   (=> [:new initial-list]
       ![object
          (state [stored-list := initial-list])
          (script
           (=> [:get-all]
               (dolist (elm stored-list) !elm)
               !:finished))]))]
```

Notice that when a sorted list object is created by the object `create-sorted-list`, the variable binding of `initial-list` is copied and then attached to the sorted list object. This variable is called an *environment variable* (see Section 7.5) of the sorted list.

Notice in this example that the object `merger` can possibly send back a partial result before the sorted-list objects have finished their tasks.

4.4 Multicast

The next example includes *multicast*. The object `alist-data-base` accepts two kinds of requests. One for registering a key and its associated value and another for specifying a key and retrieving the associated value. This object maintains a list consisting of objects each of which is created by the `create-key-value-pair` object and represents a *key-value* pair. Upon acceptance of a retrieval request, the `alist-data-base` object multicasts the specified key to all the *key-value* pair objects with the current reply destination. Each key-value pair object that has the specified key returns its value.

```
[object create-key-value-pair
  (script
   (=> [:new my-key my-value]
       ![object
          (script
           (=> [:inquire specified-key]
               (if (eq my-key specified-key) !my-value)))])))]

[object alist-data-base
  (state [alist := nil])
  (script
   (=> [:register key value]
       (temporary
        [key-value-pair :=
          [create-key-value-pair <== [:new key value]]])
       (push key-value-pair alist))
   (=> [:get key] @ reply-to
       [alist <= [:inquire key] @ reply-to]))]
```

A key-value pair object has two environment variables (see Section 7.5) `my-key` and `my-value`.

4.5 Message Passing in the Express Mode

Syntax of message passing forms in the express mode is as follows:

[*target* `<<=` *message*] or [*target* `<<=` *message* `@` *reply-destination*]	(past)
[*target* `<<==` *message*]	(now)
[*target* `<<=` *message* `$` *future-object*]	(future)

The following `clock` object receives a `[:what-time]` message sent in the express mode.

```
[object clock
  (state [time := 0])
  (script
   (=> [:start]
       (loop
        (sleep 1)
        [time := (1+ time)]))
   (=>> [:what-time]
        !time))]
```

By receiving a `[:start]` message, the object `clock` starts to tick. `(sleep 1)` is a Common Lisp form which sleeps for one second[16]. In this case, the assignment form:

```
[time := (1+ time)]
```

is executed indefinitely many times. However, if a `[:what-time]` message in the *express* mode arrives during execution of the loop form, the execution is suspended and then the current value of the state variable `time` is returned. After that, the execution of the loop form will be resumed. Notice that, if the message pattern `[:what-time]` followed `=>` instead of `=>>`, a `[:what-time]` message arriving later than a `[:start]` message would never be processed since the actions in response to the `[:start]` message will never be terminated.

4.6 Atomic Forms

The following `clock` object uses two state variables `minutes` and `seconds` instead of a single variable `time`. After accepting a `[:start]` message, this object increments the state variable `minutes` every 60 seconds.

```
[object clock
  (state
   [seconds := 0]
```

[16] In the current implementation of ABCL/1, the function `sleep` stops not only the object `clock` but also the whole ABCL/1 system.

```
 [minutes := 0])
(script
 (=> [:start]
     (loop
      (sleep 1)
      [seconds := (1+ seconds)]
      (if (= seconds 60)
          (progn
            [seconds := 0]
            [minutes := (1+ minutes)])))))
 (=>> [:what-time]
      ![minutes seconds]))]
```

However, this object is erroneous: it returns an incorrect answer to an express message `[:what-time]` if the message arrives at the very time when the object has just set `seconds` to `0` and has not yet incremented `minutes`.

In order to avoid such situations, we can specify a sequence of actions as *atomic*. When an express message arrives at an object which is executing an atomic sequence of actions, the object postpones the express message execution until these actions are completed.

The next one is a correct definition of the clock object.

```
[object clock
  (state
   [seconds := 0]
   [minutes := 0])
  (script
   (=> [:start]
       (loop
        (sleep 1)
        (atomic
         [seconds := (1+ seconds)]
         (if (= seconds 60) then
             [seconds := 0]
             [minutes := (1+ minutes)])))))
   (=>> [:what-time]
        ![minutes seconds]))]
```

In general, the following atomic form:

(atomic *form* ⋯ *form*)

specifies that the sequence of the *form*s is atomic.

4.7 Non-Resume Forms

There are often cases where a subsequently arriving *express* message should abort the currently processed ordinary message. In order to discard the suspended ordinary message execution, a *non-resume* form is available.

In response to a `[:start]` message, the following object ticks forever. However, if a `[:stop]` message in the express mode arrives, its ticking is once suspended and then aborted by the execution of the `(non-resume)` form.

```
[object clock
  (state [time := 0])
  (script
   (=> [:start]
       (loop
        (sleep 1)
        [time := (1+ time)]))
   (=>> [:what-time]
        !time)
   (=>> [:stop]
        (non-resume)))]
```

4.8 Parallel Message Passing Forms

A parallel message passing form:

$$\{\textit{message-passing-form} \cdots \textit{message-passing-form}\}$$

executes all the *message-passing-forms* in parallel and waits until the replies to all the now type messages transmitted by the form arrive. The value of this form is the list of the values of the *message-passing-forms*.

The next example includes the definition of objects which calculate the factorial of a non-negative integer.

```
[object fact
  (state [range-product := [create-range-product <== [:new]]])
  (script
   (=> [:fact n] @ reply-to
       [range-product <= [:range-from 1 :to n] @ reply-to]))]

[object create-range-product
  (script
   (=> [:new]
       ![object
```

```
(state [children :=
         {[create-range-product <== [:new]]
          [create-range-product <== [:new]]}])
(script
 (=> [:range-from i :to j]
     !(if (= i j) i
          (let ((mid (truncate (+ i j) 2)))
            (apply #'*
                   {[(first children) <==
                     [:range-from i :to mid]]
                    [(second children) <==
                     [:range-from (1+ mid) :to j]]}
            )))))])
```

The object `fact` receives a non-negative integer n and sends 1 and n, which represent the range 1 through n, to the range-product object assigned to its state variable. Each range-product object, which is created by the object `create-range-product`, receives a range i through j and sends back the product $\prod_{k=i}^{j} k$. For this purpose, a range-product object employs the following strategy: if $i = j$ is satisfied, just sending back i; otherwise,

1. dividing the range into two subranges i through $\lfloor (i+j)/2 \rfloor$ and $\lfloor (i+j)/2 \rfloor + 1$ through j,
2. sending them to the child range-product objects,
3. multiplying the sub-results from its children, and
4. sending back the result.

Notice that the state variables of an object are initialized when the first message arrives at the object (see Section 7.2). Therefore, in this example, the state variable `children` in a range-product is initialized in a lazy manner. If this state variable were initialized when the object is created, an indefinite number of range-product objects would be created.

Parallel message passing including now type message passing can be used for a simple task control. Suppose that there are four tasks A, B, C, and D and that they must satisfy the following temporal dependency condition among them (Figure 5):

1. Before B and C start, A must terminate.
2. Before D starts, B and C must terminate.

In this case, the definition of a task control object can be as follows:

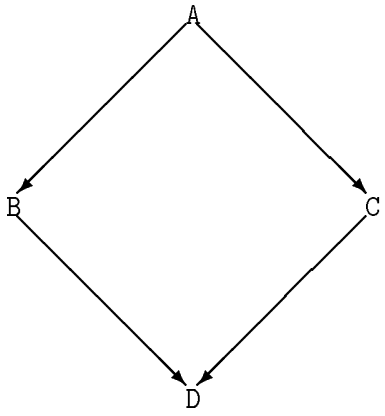

Figure 5: A Task Dependency Graph

```
[object task-controller
  (state ···)
  (script
   (=> [:start ···]
               ⋮
       ··· [task-A <== [:start]] ···
       ··· {[task-B <== [:start]] [task-C <== [:start]]} ···
       ··· [task-D <== [:start]] ···
               ⋮
       )
       ⋮
  )]
```

In a more complicated situation, we have to explicitly describe the synchronization condition in terms of future type message passing and *ready?* and/or *next-value* forms.

5 Control Structures

5.1 Common Lisp Control Structures

In ABCL/1, most imperative control structures are described in terms of Common Lisp macros and special forms. As is mentioned in Section 3.4, we can use most of them as building blocks of ABCL/1 programs. For instance, in the definition of the `merger` object in Section 4.3, we use a *block* special form, *loop* macros, a *cond* macro,

and *return-from* special forms. Also we use *loop* macros to describe the behavior of the `clock` objects in Sections 4.5 – 4.7.

The following is another program example in which Common Lisp control structures `do`, `cond`, and `if` are effectively used[17].

```
[object tree2leaves
  (script
   (=> tree
       (do ((tree tree))
           ((null tree) !:finished)
         (let ((head (pop tree)))
           (cond ((consp head)
                  (let ((head1 (pop head)))
                    (if head (push head tree))
                    (push head1 tree)))
                 (t !head))))))]
```

By the following message passing:

```
[tree2leaves <= tree @ printer]
```

the object `tree2leaves` receives a *tree* represented as a nested list structure and then sends each leaf (*i.e.*, atomic element) of the tree one by one in the left-to-right order to `printer`.

Notice that, when a now type message *tree* is sent to `tree2leaves` by the following message passing form:

```
[tree2leaves <== tree]
```

the sender object of this message receives only the first reply (see Section 4.1).

5.2 Wait-For Forms

An object which executes a *wait-for* form:

```
(wait-for
   (=> message-pattern @ reply-destination-variable
                       from sender-variable
                       where constraint
       (temporary [temporary-variable := initial-value] ···)
       behavior-description)
```

[17] Those who are familiar with Lisp may consider that this program is not efficient since unnecessary cons cells will be used during execution. We will show an improved version later in Section 5.6.

```
                ⋮
    (=> message-pattern @ reply-destination-variable
                        from sender-variable
                        where constraint
        (temporary [temporary-variable := initial-value] ···)
        behavior-description))
```

enters the waiting mode, where the object accepts a message which matches a *message pattern* specified by this form and satisfies the corresponding *constraint*. After receiving an acceptable message, the object executes the corresponding *behavior-description* part. Note that this acceptable message may have already arrived and been in the message queue when the object enters the waiting mode.

In the waiting mode, if an arriving message matches no specified message pattern, the object does not process it and it remains in the *message queue*. Notice that the object does not discard this message and will possibly process it later when the object becomes *dormant* or enters another *waiting* mode. (In contrast, when an object is in the dormant mode, if the first arriving message does not match any specified message pattern, the message will be discarded.)

The following example is a window controller object, which usually accepts *write* requests and sends them to the window under the control of this object. However, by receiving a *lock* request, the object is locked with a *key* such that no write request is processed until the lock is released. In order to release the lock, one who knows the key must send a *release* message. We assume the existence of the object `create-window` creating and returning a window object, which accepts `[:write` *character*`]` messages.

```
[object window-controller
  (state [window := [create-window <== [:new]]])
  (script
   (=> [:write character]
       [window <= [:write character]])
   (=> [:lock-with key]
       !:locked
       (wait-for
        (=> [:release-lock-with k] where (= k key)
            !:unlocked))))]
```

5.3 Wait-For-Loop Forms

A *wait-for-loop* form is in a similar syntax to a *wait-for* form:

```
(wait-for-loop
   (=> message-pattern @ reply-destination-variable
```

```
                    from sender-variable
                    where constraint
        (temporary [temporary-variable := initial-value] ···)
        behavior-description)
              ⋮
    (=> message-pattern @ reply-destination-variable
                    from sender-variable
                    where constraint
        (temporary [temporary-variable := initial-value] ···)
        behavior-description))
```

The only difference from wait-for forms is that a wait-for-loop form is executed repeatedly. In order to stop execution of a wait-for-loop form, a *return* form is available.

The following spooler object implements a spooling mechanism.

```
[object spooler
  (state
   [printer :=
     [object
       (script
        (=> any-message
            (format t "~A~%" any-message)))]])
  (script
   (=> [:start-printing] from current-client
       (wait-for-loop
        (=> [:end] from client
                   where (eq client current-client)
            (return))
        (=> [:print message] from client
                            where (eq client current-client)
            [printer <= message])))))]
```

A print request to this spooler object consists of the following sequence of messages:

`[:start-printing]`, `[:print ···]`, ···, `[:print ···]`, and `[:end]`

The sender variable `current-client` is bound to the sender object of a `[:start--printing]` message. During execution of this message, the spooler object enters the waiting mode where messages from the same sender are exclusively processed. The execution of the wait-for-loop form is not completed until an `[:end]` message from the same sender is accepted.

In this example, (the reference of) the sender object is effectively used as an ID of a spooling request. Two references $r1$ and $r2$ point the same object if and only if `(eq` $r1$ $r2$`)` returns `t`.

5.4 Match Forms

A *match* form in the following syntax:

```
(match value-form
  (is pattern_1 where constraint_1 form ...)
                    ⋮
  (is pattern_n where constraint_n form ...)
  (otherwise form ...))
```

supports the pattern matching mechanism of ABCL/1. First, the *value-form*, which can be an arbitrary form, is evaluated. If the evaluation result matches the $pattern_1$ and the $constraint_1$ is satisfied, the following *form*s are executed. Otherwise, the $pattern_2$ and $constraint_2$ are examined, and so on. If there is no pattern which matches the value of the *value-form* and whose corresponding constraint is satisfied, the forms following the *otherwise* clause are executed. Notice that the constraint parts and the otherwise clause are optional. Also notice that a pattern which is valid as a message pattern (see Section 3) is always valid as a pattern in a match form and vice versa.

In the following object definition, *match* forms are used.

```
[object create-stack
  (script
   (=> [:new]
       ![object
          (state [stack := nil])
          (script
           (=> [:push x]
               [stack := [x . stack]])
           (=> [:pop]
               (match stack
                 (is nil !:empty)
                 (is [first . rest]
                     !first
                     [stack := rest])))
           (=> [:top]
               (match stack
                 (is nil !:empty)
                 (is [first . rest]
                     !first)))
           (=> [:reset]
               [stack := nil]))])))]
```

In response to a `[:new]` message, this `create-stack` object creates and returns a stack object. The only difference of this `create-stack` object from the one in Section 3.3 is that, when a stack object created by the former is empty, it returns `:empty` as a reply to either a `[:pop]` or a `[:top]` message.

5.5 Match-Loop Forms

The syntax of a *match-loop* form is quite similar to a *match* form:

```
(match-loop value-form
  (is pattern_1 where constraint_1 form ···)
                    ⋮
  (is pattern_n where constraint_n form ···)
  (otherwise form ···))
```

If the *otherwise clause* is not omitted, a match-loop form is a repeatedly executed match form. Otherwise (*i.e.*, if the otherwise clause is omitted), a match-loop form is executed repeatedly until no pair of a pattern and a constraint is satisfied by the value of the *value-form*. In both cases, a *return* form can stop execution of the match-loop form.

The following object `reverse` successively receives a sequence of messages enclosed by `[:start]` and `[:end]` messages. Each message except the first one (`[:start]`) and the last one (`[:end]`) represents a page image to be printed by a laser beam printer. In response to this sequence of messages, this object sends the page images in the reverse order to the object `laser-beam-printer`.

```
[object reverse
  (state [stack := [create-stack <== [:new]]])
  (script
   (=> [:start]
       [stack <= [:reset]])
   (=> [:page image]
       [stack <= [:push image]])
   (=> [:end]
       (match-loop [stack <== [:pop]]
         (is :empty
             (return))
         (is page-image
             [laser-beam-printer <= [:page page-image]])))))]
```

5.6 Routine Calls

A *routine call* is a form in the following syntax:

(*routine-name argument* ···)

The *routine-name* is the name of a routine defined in the routine declaration part. The *argument*s are passed to the routine in a *call-by-value* manner.

The following object `tree2leaves` contains routine calls.

```
[object tree2leaves
  (script
   (=> tree
       (extract tree)
       !:finished))
  (routine
   (extract (tree)
     (match tree
       (is [[A . B] . C]
           (extract [A . B])
           (extract C))
       (is [A . B]
           !A
           (extract B)))))]
```

This definition is a revised version of the one in Section 5.1. The routine `extract` is defined recursively and takes a single argument `tree`.

Routine calls can be used for the purpose of code sharing. The next one is such an example:

```
[object clock
  (state
   [seconds := 0]
   [minutes := 0])
  (script
   (=> [:start]
       (loop
        (sleep 1)
        (atomic
         [seconds := (1+ seconds)]
         (if (= seconds 60)
             (progn
               [seconds := 0]
               [minutes := (1+ minutes)])))))
    (=>> [:what-time]
         ![minutes seconds])
```

```
    (=>> [:reset]
         (set-time 0 0))
    (=>> [:reset-and-stop]
         (set-time 0 0)
         (non-resume))
    (=>> [:set m s]
         (set-time m s)))
  (routine
    (set-time (m s)
      [minutes := m]
      [seconds := s]
      [printer <=
        (format nil
                "Set the current time to (~D ~D)." m s)]))]
```

The routine `set-time` is called just after a `[:reset]`, `[:reset-and-stop]`, or `[:set-time ···]` message in the express mode arrives. Since this routine is only called during express message execution, we need not use any atomic form in the body of `set-time`.

For the purpose of non-local exit from a routine, a *return-from* special form:

`(return-from` *routine-name form*`)`

of Common Lisp is available. The return-from form stops execution of the routine (named by the *routine-name*) and lets the routine return the evaluation result of the *form*. Notice that the *form* is optional and its default value is `nil`.

A routine call can explicitly return *multiple values* using the Common Lisp function *values*.

5.7 Defining Macros

The user can define his or her own *macros* in ABCL/1. For this purpose, a *defmacro* form is available. The following example includes a *defmacro* form, which defines the macro `wait-for-loop-macro` equivalent to the ABCL/1 control structure `wait-for-loop` (see Section 5.2).

```
(defmacro wait-for-loop-macro (&body body)
  `(loop (wait-for ,@body)))
```

In the above example,

`(&body body)`

means that the formal parameter `body` will be bound to the list of all the arguments of a macro call and:

```
`(loop (wait-for ,@body)))
```

is an abbreviation of:

```
(list 'loop (cons 'wait-for body))
```

The macro `wait-for-loop-macro` expands the following macro call:

```
(wait-for-loop-macro
 (=> [:end] from client
            where (eq client current-client)
     (return))
 (=> [:print message] from client
                     where (eq client current-client)
     [printer <= message]))
```

into the form[18]:

```
(loop
 (wait-for
  (=> [:end] from client
             where (eq client current-client)
      (return))
  (=> [:print message] from client
                      where (eq client current-client)
      [printer <= message])))
```

Note that a macro definition must be loaded into the ABCL/1 system before an object where the macro occurs is loaded. Therefore, *defmacro* forms should appear on the head parts of ABCL/1 source files. Note also that, during expansion of a macro call, no ABCL/1 specific forms (*e.g.*, message passing forms) except *match* and *match-loop* forms can be called. In contrast, the expanded form can include ABCL/1 specific forms.

The next example defines the `psend` macro which simulates a simple case of parallel message passing (see Section 4.8). This macro does not assume any *multicast* as its argument.

```
(defmacro psend (&body body)
  (let (message-passing-forms
        next-value-forms
```

[18]This form is also a macro call since *loop* is a macro of Common Lisp.

```
      make-future-forms
      (temp1 (gensym))
      (temp2 (gensym)))
  (dolist (mess-pass body)
    (match mess-pass
      (is `[,target <= ,message]
          (push mess-pass message-passing-forms)
          (push nil next-value-forms))
      (is `[,target <= ,message ,at-or-dollar ,reply-to]
          where (member at-or-dollar '($ @))
          (push mess-pass message-passing-forms)
          (push nil next-value-forms))
      (is `[,target <== ,message]
          (push '(make-future) make-future-forms)
          (let ((target (second mess-pass))
                (message (fourth mess-pass)))
            (push `[,target <= ,message $ (pop ,temp1)]
                  message-passing-forms))
          (push `(next-value (pop ,temp2))
                next-value-forms))))
  (setq make-future-forms (nreverse make-future-forms))
  (setq message-passing-forms (nreverse message-passing-forms))
  (setq next-value-forms (nreverse next-value-forms))
  `(let* ((,temp1 (list ,@make-future-forms))
          (,temp2 ,temp1))
    ,@message-passing-forms
    (list ,@next-value-forms))))
```

We do not explain the details of this definition[19]. We just show an example. The following macro call:

```
(psend [O1 <= M1 $ f] [O2 <== M2] [O3 <== M3] [O4 <= M4])
```

is expanded to the form:

[19]However, those who are interested in this definition should notice that, under the current implementation, a pattern:

$$\texttt{`[},variable_1\ \texttt{,}variable_2\ \cdots\ \texttt{,}variable_n\texttt{]}$$

matches a form enclosed with brackets:

$$\texttt{[}form_1\ form_2\ \cdots\ form_n\texttt{]}$$

and after this matching the pattern variable $variable_i$ is bound to $form_i$.

```
(let* ((g1734 (list (make-future) (make-future)))
       (g1735 g1734))
  [O1 <= M1 $ f]
  [O2 <= M2 $ (pop g1734)]
  [O3 <= M3 $ (pop g1734)]
  [O4 <= M4]
  (list nil
        (next-value (pop g1735))
        (next-value (pop g1735))
        nil))
```

where `g1734` and `g1735` are uninterned (*i.e.*, fresh and unique) symbols created by the Common Lisp function *gensym*.

6 Miscellaneous Forms

6.1 Suicide

An ABCL/1 object can kill itself. For this purpose a *suicide* form is available. After receiving a `[:die]` message in the express mode and executing the suicide form, the following clock object becomes dead and accepts no more messages.

```
[object clock
  (state [time := 0])
  (script
   (=> [:start]
       (loop
        (sleep 1)
        [time := (1+ time)]))
   (=>> [:what-time]
        !time)
   (=>> [:die]
        (suicide)))]
```

When a message arrives at a dead object, the current ABCL/1 system prints a warning and invokes the ABCL/1 inspector (see Section 13).

6.2 Break Points

Once some object executes a *break-point* form:

`(break-point)` or `(break-point` *string*`)`

the execution of all the active objects are suspended and, if the *string* argument is supplied, it is printed. Then the ABCL/1 inspector starts working on the object in which the break-point form is executed. The detail of the ABCL/1 inspector will be described in Section 13.

7 Variables and Their Scope Rule

The ABCL/1 employs the lexical scope rule. The variables in ABCL/1 are divided into the following sorts:

- Variables in the global environment
- State variables
- Temporary variables
- Pattern variables
- Environment variables
- Pseudo variables

Temporary and pattern variables are often collectively called *local* variables[20].

Every variable that is not in the global environment belongs to just a single object. Therefore, when an object is created, its own (or distinct) variables are created for the object: no such variables are shared by other objects.

7.1 Variables in the Global Environment

Each variable in the global environment must be created and assigned at the *top level*, which is the shell of the ABCL/1 system and can be regarded as a special object (see Section 8.1). At the top level, variables and constants can be declared using *defconstant*, *defvar*, and *defparameter* forms of Common Lisp. Each object can refer to a variable in the global environment by its name but no object (except the top level shell object) can update the contents of any variable in the global environment.

Each object with a global name *object-name* is the object to which the variable *object-name* in the global environment is bound. This variable binding must be established at the top level. For instance, executing the following form:

[20] This does not mean that the temporary and pattern variables of an object are the only locally defined variables within the object. The term "local" is used in the sense that the scope of each temporary and pattern variable is a local lexical block within an object

```
[object A
  (state ···)
  (script ···)]
```

the top level creates an object satisfying this object definition form and binds the variable `A` in the global environment to the object. Every object can refer to the object by its name (*i.e.*, the name `A` of the variable in the global environment).

7.2 State Variables

A *state variable* is declared in the state variable declaration part (see Section 2) of an object definition form. Every form executed by an object may inquire/update the contents of its state variables.

No state variables of an object are initialized at the birth time of the object. Instead, at the birth time of an object, its state variables are assigned to `nil`. The initialization procedure is invoked when the first message to the object arrives.

The following example depends on the fact that the initialization procedure of state variables of an object is invoked when the first message arrives at the object. This example consists of the definitions of two objects, which co-operate and generate prime numbers. The object `generator` generates an increasing sequence of integers from 2 as candidates for prime numbers and sends them to a filter object. Each filter object created by the `create-filter` stores some prime number in its state variable `my-number` and filters out multiples of the prime number.

```
[object generator
  (state
   [filter := [create-filter <== [:new]]]
   [n := 1])
  (script
   (=> [:generate]
       (loop
         [filter <= [:check (incf n)]])))]

[object create-filter
  (script
   (=> [:new]
       ![object filter
          (state [next-filter := [create-filter <== [:new]]])
          (script
           (=> [:check n]
               (print n)
               (wait-for-loop
```

```
(=> [:check m]
    (unless (zerop (mod m n))
            [next-filter <= [:check m]])))))]))]
```

Notice that, in this example, filter objects are created in a lazy manner. Otherwise, no prime number could be obtained and filter objects would be kept created forever.

7.3 Temporary Variables

A *temporary* variable is a local variable whose scope is some lexical block of an object. Temporary variable is declared either in a form:

(temporary [*temporary-variable* := *initial-value*] ···)

or in a *let* or *let** special form of Common Lisp. In the former case, the scope of the variable is the behavior description part following the *temporary* variable declaration. In the latter case, *let* and *let** special forms have the same scope as those in Common Lisp[21].

Since ABCL/1 employs the static scope rule, in the following example, the temporary variable `x` declared following the message pattern `[:start]` cannot be accessed from the behavior description of the routine `f`. Also, even if an express message `[:interrupt]` arrives while an ordinary message `[:start]` is being executed, the object can neither inquire nor update the contents of this temporary variable `x` during the execution of the express message.

```
[object A
  (script
   (=> [:start]
       (temporary [x := nil])
       ⋮
       (f)
       ⋮      )
   (=>> [:interrupt]
        ⋮      ))
  (routine
   (f ··· ))]
```

[21] However, each temporary variable in ABCL/1 has dynamic extent, whereas, in Common Lisp each variable declared by a let or let* form has indefinite extent[?].

7.4 Pattern Variables

Pattern variables fall into two sorts:

1. elements of message patterns including reply destination variables and sender variables

2. elements of patterns in match and match-loop forms

In case of 1, the scope of a pattern variable is the constraint, temporary variable declaration, and behavior description part following the message pattern it belongs to. In case of 2, the scope of a pattern variable is the forms following the message pattern it belongs to.

Every time a pattern is being matched against some value, the pattern variables in the pattern are bound to the corresponding values. These variables cannot be re-bound within its scope. This fact is one of the most significant differences between pattern variables and state/temporary variables.

7.5 Environment Variables

In the following example, when the object `A` accepts a `[:new ···]` message, an object satisfying the inner object definition form following the exclamation mark (`!`) is created.

```
[object A
  (state [x := nil])
  (script
   (=> [:new y]
       ![object ··· x ··· y ···])
          ⋮
    )]
```

In this case, the variable bindings for `x` and `y` are copied and then attached to the environment of the created object. Therefore, this object has its own variables whose names are `x` and `y`, which are called *environment variables*.

In general, when an object `O` is created by another one `C`, the bindings of the variables of `C` which can be accessed at that time are copied and then attached to the environment of `O`. The variables of `O` created in this manner are called its *environment variables*.

An object can read the contents of its environment variables but it cannot update the contents.

7.6 Pseudo Variables

Each object has a pseudo variable `Me` which is always bound to the object itself. `Me` is treated as if it were an environment variable, that is, its contents can be read but cannot be updated by the object.

The following example contains the use of `Me`.

```
[object create-clock
  (script
   (=> [:new]
       ![object
          (state [time := 0])
          (script
           (=> [:start]
               (sleep 1)
               [time := (1+ time)]
               [Me <= [:start]])
           (=>> [:what-time]
                !time))]))]
```

The object `create-clock` creates a clock object, which sends a past type message to `Me` (*i.e.*, to itself) in order to invoke itself again and again.

PART II

8 The ABCL/1 System

ABCL/1 is not only an object-oriented concurrent language but also an interactive programming system. Through the following sessions, we will illustrate how to interact with the ABCL/1 system.

The ABCL/1 system is currently running on Sun-3 and Sun-4 workstations and Symbolics Lisp machines. The ABCL/1 system on Sun workstations is implemented in KCl (Kyoto Common Lisp) and the one on Symbolics lisp machines is implemented in Symbolics Common Lisp. Both versions were implemented by Y. Ichisugi. The following session examples are based on the KCl version of the ABCL/1 system.

The current ABCL/1 system is implemented on a single processor machine and ABCL/1 programs are executed in a pseudo parallel manner[22]. The ABCL/1 system maintains a *scheduling queue*, in which objects in the active mode are stored, and employs the round-robin style scheduling policy.

8.1 Invocation of the ABCL/1 System

In order to invoke the ABCL/1 system on a Sun workstation, execute the shell command `abcl1`.

```
----------------------------------------
% abcl1

       ____       _____       ______      __                  ___    TM
 NEW  /    \     |  __ \     |      |    |  |           //   |_  |
     |  /\  |    | |__| |    |   ---+    |  |          //     |  |
     | |__| |    |  __ <     |  |        |  |         //      |  |
     |  __  |    | |__| |    |   ---+    |  +---+    //       |  |
     |_|  |_|    |_____/     |______|    |______|   //        |__|

                                                  ⋮
                                                  ⋮

<ABCL/1> □
----------------------------------------
```

[22] We have a prototype version of a multi-computer implementation[?].

The top level, which is a shell of the ABCL/1 system, prompts `<ABCL/1>`. In ABCL/1, the top level is a special object, which interacts with the user.

In the subsequent session examples, the symbol □ indicates the current cursor position.

8.2 Halting the ABCL/1 System

By typing a form `(bye)` or `(by)`, we can halt the ABCL/1 system and return to the Unix shell.

```
---------------------------------------
<ABCL/1> (bye)
Bye.
% □                    ;; Returning to Unix.
---------------------------------------
```

9 The Top Level

9.1 The Help Command

The top level prints the following help message in response to the help command `?`.

```
---------------------------------------
<ABCL/1> ?

    The following commands are available:
    (bye) or (by)              --  Exit from ABCL/1
    (inc "<filename>" ...)     --  Include ABCL/1 source or binary ···
    (comp "<filename>" ...)    --  Compile ABCL/1 source files
    (show-objects)             --  List up objects defined at toplevel
             ⋮
<ABCL/1> □
---------------------------------------
```

9.2 Object Definition at the Top Level

At the top level, the user can define an object by just typing its definition form.

```
---------------------------------------
<ABCL/1> [object counter
           (state [c := 0])
```

```
            (script
             (=> [:add n] where (and (integerp n) (plusp n))
                 [c := (+ c n)])
             (=> [:value]
                 !c)
             (=> [:reset]
                 [c := 0]))]

<ABCL/1> □
---------------------------------------
```

An object whose global name is `counter` is defined.

9.3 Evaluation of Forms at the Top Level

The user can obtain the evaluation result of a form by just typing the form at the top level. For instance, the user can send a message from the top level to the object `counter` as follows:

```
---------------------------------------
<ABCL/1> [counter <== [:value]]
0               ;; The initial value is 0.

<ABCL/1> [counter <= [:add 3]]

<ABCL/1> [counter <= [:add 2]]

<ABCL/1> [counter <== [:value]]
5               ;; The current value is 5.

<ABCL/1> □
---------------------------------------
```

In response to the first `[:value]` message, the object `counter` returns the value 0 to the top level. The returned value 0 is printed on the screen. On reception of the second and third messages, the object `counter` increments its contents by 3 and 2, respectively. In these cases, the top level does not receive any reply and, thus, nothing is printed on the screen. The last message `[:value]` lets `counter` return its contents (i.e., 5) to the top level.

The following is an execution example of a *setq* form.

```
---------------------------------------
```

```
<ABCL/1> (setq x [counter <== [:value]])
5

<ABCL/1> x
5

<ABCL/1> □
----------------------------------------
```

By execution of the above setq form, the variable `x` in the global environment is dynamically created and the returned value 5 is assigned to this variable. When the user types the name of a variable, the top level prints the current value of the variable.

The top level can accept more complicated forms.

```
----------------------------------------

<ABCL/1> (dolist (message (list [:reset] [:add 3] [:add 2]))
           [counter <= message])
nil

<ABCL/1> [counter <== [:value]]
5

<ABCL/1> □
----------------------------------------
```

Each time the top level reads a form, it creates and activates an object, which evaluates the form and prints the evaluation result. The top level does not prompt until no active object remains.

10 The Loader and Compiler

10.1 Loading an ABCL/1 Program

It is often the case that the user would like to make a program with his/her favorite editor and then load it into the ABCL/1 system. For this purpose, an *inc* form is available at the top level.

Assume that the user prepares the object definition of `counter` with some editor and saves it in a file whose name is `counter.abcl1`[23]. The effects of the following two scenarios are equivalent.

[23]The current ABCL/1 system supposes that the name of every ABCL/1 source file ends with "`.abcl1`".

```
---------------------------------------
<ABCL/1> (inc "counter")
         ;;
         ;; Loading the file counter.abcl1
         ;;
<ABCL/1> □
---------------------------------------

---------------------------------------
<ABCL/1> [object counter
           (state [c := 0])
           (script
            (=> [:add n] where (and (integerp n) (plusp n))
                [c := (+ c n)])
            (=> [:value]
                !c)
            (=> [:reset]
                [c := 0]))]

<ABCL/1> □
---------------------------------------
```

In general, *inc* forms are in the following syntax:

(`inc` *file-name* ··· *file-name*)

The specified files are loaded in the left to right order. When a *file-name* ends with neither `.abcl1`, `.lsp`, nor `.o`, the latest file among the source code file *file-name*`.abcl1`, the intermediate code file *file-name*`.lsp`, and the compiled code file *file-name*`.o` is loaded into the ABCL/1 system.

Note that when an *inc* form loads an ABCL/1 source file *file-name*`.abcl1`, it automatically creates the intermediate code file *file-name*`.lisp`.

10.2 Compiling an ABCL/1 Program

In order to compile an ABCL/1 source file (or an intermediate code file), a *comp* form in the following syntax:

(`comp` *file-name* ··· *file-name*)

is available. The above form compiles the specified files. Compiled files can be loaded using *inc* forms.

11 Getting Information about Objects

11.1 Listing the Objects Defined at the Top Level

A *show-objects* form prints the list of the objects defined at the top level.

```
---------------------------------------
<ABCL/1> (show-objects)

*** List of objects defined at toplevel ***
        counter                  ;; counter is the only object currently
                                 ;; defined at the top level.
<ABCL/1> □
---------------------------------------
```

11.2 Describing Objects

A *describe* form prints information about the specified object.

```
---------------------------------------
<ABCL/1> (describe counter)
ABCL/1 Object #<counter 0> :
Mode: dormant                    ;; This object is in the dormant mode.

Defined Protocols        ;; Ordinary messages beginning with :add, :value,
Ordinary: (:add :value :reset)   ;; and :reset are acceptable.
Express: nil                     ;; No express messages are acceptable.
state    c        = 5            ;; The value of the state variable c is 5.

<ABCL/1> □
---------------------------------------
```

In Section 1, we say that:

> each object is in one of the three modes, *dormant*, *waiting*, and *active* at any time.

However, in the actual implementation, each object is one of the following six modes:

uninitialized
: an object in this mode has never received any message.

dormant
: an object in this mode has received at least one message but currently it does not process any message.

active
: an object in this mode is now processing a message.

wait-for-wait
: an object in this mode has entered the waiting mode by executing a wait-for(-loop) form.

value-wait
: an object in this mode has entered the waiting mode by executing a now type message passing form or a next-value form.

dead
: an object in this mode has already executed a suicide form.

The *wait-for-wait* mode and *value-wait* mode are often collectively called waiting modes.

11.3 Listing the Message Protocols of an Object

A *protocol* form prints the message protocols, the first component of each message pattern, of the specified object.

```
---------------------------------------
<ABCL/1> (protocol counter)

Defined Protocols
Ordinary: (:add :value :reset)
Express: nil

<ABCL/1> □
---------------------------------------
```

12 Resetting Objects

A *reset* form in the following syntax:

(reset *object* ··· *object*)

forces the specified objects to enter the *dormant mode*. At that time, each specified object either in a *waiting mode* (*wait-for-wait*, or *value-wait*) or in the *dead mode* aborts the current computation and the contents of its message queue are discarded. An object in the uninitialized mode cannot be reset by a reset form.

A *full-reset* form in the following syntax:

`(full-reset)` or `(full-reset` *object* $\cdots$ *object*`)`

forces the specified objects to enter the *uninitialized mode*. If this form has no argument, we consider that all the objects defined at the top level are specified. In this case, each object that is not defined at the top level becomes garbage. By this form, the values of the state variables of each specified object are set to `nil` and they will be initialized again when the next message arrives at the object.

13 The Inspector

As is mentioned above, the user can get information about objects using a *describe* form at the top level. More information about them can be obtained from the *inspector*, which is invoked by an *inspect* form. The inspector can also be invoked by a *break-point* form (see Section 14.3).

In order to inspect the object `counter`, do as follows:

```
----------------------------------------
<ABCL/1> (inspect counter)

<INSPECTING #<counter 0>> □
----------------------------------------
```

In this example, `#<counter 0>` is the *print-name* of the object which is the first one among those whose global or local name is `counter`. Note that the print-name of the $n+1$-th object whose global or local name is *object-name* is `#<`*object-name* *n*`>`.

The inspector accepts almost all forms available at the top level but no message transmission is allowed from the inspector. In the inspector, a form is evaluated in the environment of the inspected object.

```
----------------------------------------
<INSPECTING #<counter 0>> c
                                   ;; The value of the (state) variable c
5                                  ;; of the object counter is 5.
----------------------------------------
```

13.1 The Help Command in the Inspector

The inspector accepts the help command `?` and prints the following help message.

```
----------------------------------------
<INSPECTING #<counter 0>> ?
Almost all top-level commands are available.
```

```
Also the following commands are available:
        ?               list up commands
        :man <com>      help for <com>
        :stat           status of the object
        :where          indicate current location of execution counter
        :proto          acceptable message patterns
                        .
                        .
                        .
                        .
<INSPECTING #<counter 0>> □
---------------------------------------
```

13.2 Listing Information about the Inspected Object

In the inspector:

- the command `:stat` prints the current mode and the current values of the state/environment variables of the inspected object,
- the command `:mode` prints the current mode of the object,
- the `:proto` command prints the message protocols of the object,
- the command `:l` prints the names of the variables in the current environment[24] of the object,
- the command `:v` prints the current values of the variables in the current environment of the object,
- the commands `:ls`, `:le`, and `:ll` print the names of the state, environment, and local (*i.e.*, temporary and pattern) variables, respectively.
- the commands `:vs`, `:ve`, and `:vl` prints the values of the state, environment, and local variables, respectively.

In case of the `:l` command, the name of the object is also listed as a constant.

```
---------------------------------------
<INSPECTING #<counter 0>> :stat    ;; Printing the current status:
Mode: dormant                      ;; the current mode is dormant;
```

[24] In general, the inspected object may be in the active mode (see Section 13.4, 14.3) and suspended by the ABCL/1 system. The current environment of an object depends on the lexical location on which the execution is suspended.

```
state    c        = 5              ;; the value of the state variable c is 5;
                                   ;; this object has no environment variables.
<INSPECTING #<counter 0>> :proto   ;; Listing the message protocols.

Defined Protocols
Ordinary: (:add :value :reset)
Express: nil

<INSPECTING #<counter 0>> :mode    ;; Printing the current mode.
Mode: dormant

<INSPECTING #<counter 0>> :l       ;; Listing the variable names:
constant counter                   ;; counter is the only constant;
state   c                          ;; c is the only state variable.

<INSPECTING #<counter 0>> :v       ;; Listing the values of the variables:
constant counter  = #<counter 0>   ;; the value of counter is an object;
state    c        = 5              ;; the value of the state variable c is 5.

<INSPECTING #<counter 0>> :vs      ;; Listing the values of the state variables.
state    c        = 5

<INSPECTING #<counter 0>> □
---------------------------------------
```

13.3 Halting the Inspector

By the inspector command `:q`, control returns to the top level.

```
---------------------------------------
<INSPECTING #<counter 0>> :q
Reset #<top-level 0> to DORMANT mode.

<ABCL/1> [counter <== [:value]]
5

<ABCL/1> □
---------------------------------------
```

When the inspector is invoked during execution of objects (see Section 13.4, 14.3), the command `:q` forces all the active objects to enter the dormant mode. In other words, the inspector *reset*s (see Section 12) these objects. If the user wants to resume

the execution, s/he should use the command `:c` (see Section 13.7) instead of the command `:q`.

13.4 Interruption from the Keyboard

During execution of objects, pressing the <return> key suspends the execution of all the objects and invokes the inspector on the object which is being executed just when the interruption occurs[25]. This mechanism is useful when objects enter infinite computation.

Assume that the following object definition forms are stored in the file whose name is `prime.abcl1`.

```
[object generator
  (state
   [filter := [create-filter <== [:new]]]
   [n := 1])
  (script
   (=> [:generate]
       (loop
        [filter <= [:check (incf n)]])))]

[object create-filter
  (script
   (=> [:new]
       ![object filter
          (state [next-filter := [create-filter <== [:new]]])
          (script
           (=> [:check n]
               (print n)
               (wait-for-loop
                (=> [:check m]
                    (if (not (zerop (mod m n)))
                        [next-filter <= [:check m]])))))]))]
```

From now on, **<return>** in session examples means that the user presses the <return> key.

```
----------------------------------------
<ABCL/1> (inc "prime")
    ;;
```

[25]Since the current ABCL/1 system runs on a single processor system, a single object is being executed at a time.

```
    ;; Loading the definitions of generator and create-filter.
    ;;
<ABCL/1> [generator <= [:generate]]   ;; Starting Execution.

2
3
5
7
11
13
17
19
23
<return>
Console interrupt
Type ? for help.

<INSPECTING #<filter 1>> □
---------------------------------------
```

13.5 Recursive Invocation of the Inspector

The user sometimes wants to inspect an object which has no global name and to which some variable of another object is bound. In such a case, the user can recursively invoke the inspector with the command `:ins`. The argument of this command is a form whose evaluated result is the object to be inspected. In a recursive inspection level, the command `:ret` exits from the current level and control moves up to the previous level.

```
---------------------------------------
<INSPECTING #<filter 1>> :vs
state    next-filter = #<filter 2>

<INSPECTING #<filter 1>> :ins next-filter
                              ;; recursive invocation of the inspector.
<INSPECTING #<filter 2>> :stat
Mode: active
state    next-filter = #<filter 3>
env      create-filter = #<create-filter 0>

<INSPECTING #<filter 2>> :ins next-filter
```

```
<INSPECTING #<filter 3>> :stat
Mode: wait-for
state    next-filter = #<filter 4>
env      create-filter = #<create-filter 0>

<INSPECTING #<filter 3>> :ret
                                    ;; exiting from the current level.
<INSPECTING #<filter 2>> :ret

<INSPECTING #<filter 1>> :stat
Mode: active
state    next-filter = #<filter 2>
env      create-filter = #<create-filter 0>

<INSPECTING #<filter 1>> □
---------------------------------------
```

13.6 Listing the Active Objects

A *show-system-queue* form prints the objects in the active modes and those in the waiting modes. In the current implementation, the former ones are stored in the system scheduling queue and the latter ones are stored in the waiting object pool.

```
---------------------------------------
<INSPECTING #<filter 1>> (show-system-queue)
*** ACTIVE objects  ***
#<filter 0>
#<generator 0>
#<filter 7>
#<filter 6>
#<filter 5>
#<filter 2>
#<filter 1>

*** WAIT-FOR objects ***
#<filter 4>
#<filter 3>
#<filter 8>

*** VALUE-WAIT objects ***
```

```
nil
<INSPECTING #<filter 1>> □
--------------------------------------
```

13.7 Continuing the Suspended Computation

In order to resume the execution suspended by <return>, the inspector command `:c` is available. This command also halts the inspector.

```
--------------------------------------
<INSPECTING #<filter 1>> :c        ;; Resuming the suspended execution.

29
31
37
41
.
.
.
--------------------------------------
```

Typing `:q` instead of `:c` in this situation, the suspended computation is terminated and the top level becomes active. In this case, all the filters and the generator are reset and enter the dormant mode.

13.8 Getting the Contents of the Program Counters of Objects

Each ABCL/1 object has its own program counter, which keeps the position currently being processed. The inspector can show the contents of the program counter of the inspected object. For this purpose, the inspector command `:where` is available. Also, at the top level and in the inspector, a *where* form:

`(where` *object*`)`

prints the contents of the program counter of the specified object.

When an object is executing a routine or an express message, the contents of the program counter in the previous context(s) are pushed on the execution stack of the object. In this case, a *where* form and a *:where* command print not only the current contents of the program counter but also those of the top five and bottom five ones on the stack.

```
----------------------------------------
<INSPECTING #<filter 1>> :where

Mode: active
@("/ua/aoyagi/abcl1/prime.abcl1" 2 2 1 2 3 1 3)

<INSPECTING #<filter 1>> □
----------------------------------------
```

In this example, the first element of the list following an `@` is the name of the file from which the object definition of `#<filter 1>` is loaded. The sequence of the numbers:

```
2 2 1 2 3 1 3
```

following the file name represents the position at which the current execution is suspended. The first `2` means that the position is in the second form in the file:

```
"/ua/aoyagi/abcl1/prime.abcl1",
```

that is, the following object definition form:

```
[object create-filter
  (script
   (=> [:new]
       ![object filter
          (state [next-filter := [create-filter <== [:new]]])
          (script
           (=> [:check n]
               (print n)
               (wait-for-loop
                (=> [:check m]
                    (if (not (zerop (mod m n)))
                        [next-filter <= [:check m]])))))]))]
```

The second `2` means that the place is in the third element of the object definition form. In the following, `0` represents the first element, `1` represents the second one, `2` represents the third one, and so on.

```
                        0          1                2
(2 1 2 3 1 3)   [object  create-filter  (script ···)]
                        0          1
(1 2 3 1 3)     (script  (=> ···))
                        0          1          2
(2 3 1 3)       (=>  [:new]  ![object ···])
                        0          1          2          3
(3 1 3)         ![object  filter  (state ···)  (script ···)]
                        0          1
(1 3)           (script  (=> [:check n] ···))
                        0          1          2          3
(3)             (=>  [:check n]  (print n)  (wait-for-loop ···))
```

In the above case, the position is the wait-for-loop form.

Note that in the ABCL/1 mode of Gnu Emacs, it is easy to find the position in the source file from an output of a where form or a :where command (see Section 19).

14 Execution Monitoring

14.1 The Tracer

During execution, when some *event* (*e.g.*, message transmission, message acceptance, or mode transition) occurs on one of the specified objects, the *tracer* prints the information about the event. We say that the *trace-flag* of an object is *on* when the object is specified to be traced by the tracer.

In order to turn on the trace flag of objects, a *trace-objects* form is available whose syntax is as follows:

`(trace-objects` *object* ··· *object*`)`

When a *trace-objects* form takes no argument:

`(trace-objects)`

the trace flags of all the objects become *on*.

In order to turn off the trace flags of objects, an *untrace-objects* form is available whose syntax is as follows:

`(untrace-objects` *object* ··· *object*`)`

When an *untrace-objects* form takes no argument:

`(untrace-objects)`

the trace flags of all the objects become *off*.

The value of a *trace-objects* or *untrace-objects* form is either the list of the objects whose flags are *on* or the symbol `:trace-all-objects`.

```
---------------------------------------
<ABCL/1> (inc "prime")
    ;;
    ;; Loading the definitions of generator and create-filter.
    ;;
<ABCL/1> (show-objects)          ;; Listing the objects defined at the
                                 ;; top level.
*** List of objects defined at toplevel ***
        create-filter
        generator

<ABCL/1> (trace-objects generator)  ;; Tracing generator.
(#<generator 0>)

<ABCL/1> [generator <= [:generate]]
{Time 4  } #<generator 0>: ['#<create-filter 0> <== '(:new)]
{Time 4  } #<generator 0> became VALUE-WAIT.
{Time 9  } #<generator 0> received value '#<filter 0> and became ···
{Time 9  } #<generator 0> accepted '(:generate) from #<top-level 0>
{Time 12 } #<generator 0>: ['#<filter 0> <= '(:check 2)]
<RETURN>                    ;; Pressing the <return> key.

Console interrupt
Type ? for help.

<INSPECTING #<generator 0>> □
---------------------------------------
```

Every line printed by the tracer is in the following format:

{Time *number*} *event-description*

where *number* represents the global time of the system when the corresponding event occurs[26].

[26] In the computation model of the language ABCL/1, no global time is assumed (see Section 1). However, the current ABCL/1 system maintains the global time for ease of debugging.

14.2 Recording Event Histories

An object can record its event history including message transmissions and acceptances. A *history-on* form starts recording the event histories of all the objects and a *history-off* form stops recording them. Later on, a *history* form (at the top level) or the inspector command `:hist` displays the recorded history of the specified object.

```
---------------------------------------
<ABCL/1> (full-reset)              ;; Initializing the objects.
Reset #<create-filter 0> to UNINITIALIZED mode.
Reset #<generator 0> to UNINITIALIZED mode.
*all-abcl-objects* was reset.
<ABCL/1> (history-on)      ;; Start recording the event histories.
t

<ABCL/1> [generator <= [:generate]]

2
3
5
7
11
<RETURN>                           ;; Pressing the <return> key.
Console interrupt
Type ? for help.

<INSPECTING #<filter 4>> :hist     ;; Printing the history of the fifth filter.
{Time 49 } #<filter 4> was created by #<create-filter 0>
{Time 68 } #<filter 4>: ['#<create-filter 0> <== '(:new)]
{Time 68 } #<filter 4> became VALUE-WAIT.
{Time 77 } #<filter 4> received value '#<filter 5> and became ACTIVE
{Time 77 } #<filter 4> accepted '(:check 7) from #<filter 3>
{Time 85 } #<filter 4> became WAIT-FOR.
{Time 98 } #<filter 4> accepted '(:check 11) from #<filter 3>
{Time 102} #<filter 4>: ['#<filter 5> <= '(:check 11)]
{Time 113} #<filter 4> became WAIT-FOR.
{Time 122} #<filter 4> accepted '(:check 13) from #<filter 3>
{Time 127} #<filter 4>: ['#<filter 5> <= '(:check 13)]

<INSPECTING #<filter 4>> (history generator :start 5 :end 30)
        ;; Printing the history between time 5 and time 30 of generator.
{Time 9  } #<generator 0> received value '#<filter 1> and became ···
```

```
{Time 9  } #<generator 0> accepted '(:generate) from #<top-level 0>
{Time 12 } #<generator 0>: ['#<filter 1> <= '(:check 2)]
{Time 16 } #<generator 0>: ['#<filter 1> <= '(:check 3)]
{Time 21 } #<generator 0>: ['#<filter 1> <= '(:check 4)]
{Time 25 } #<generator 0>: ['#<filter 1> <= '(:check 5)]

nil
<INSPECTING #<filter 4>> (history-off)
                              ;; Stop recording the event histories.
nil
<INSPECTING #<filter 4>> □
----------------------------------------
```

14.3 Break Points

When some object executes a *break-point* form, the current execution of all the objects is suspended and the inspector is invoked. For instance, suppose that the object `generator` is modified as follows:

```
----------------------------------------
[object generator
  (state
   [filter := [create-filter <== [:new]]]
   [n := 1])
  (script
   (=> [:generate]
       (loop
        (if (= n 50) (break-point))
        [filter <= [:check (incf n)]])))]
----------------------------------------
```

The execution of the `generator` is suspended after transmitting integers up to 50.

The following session example shows the effects of a break-point form:

```
----------------------------------------
<ABCL/1> [generator <= [:generate]]

2
3
5
7
11
```

```
13
17
19
23
29
31
Break:
<INSPECTING #<generator 0>> □
---------------------------------------
```

Notice that, in this case, each number in the interval from 32 to 50 is still stored in some filter object or has already been filtered out.

A break-point form may take a *string* argument. In such a case, the string argument is printed when a break-point form is executed. For instance, in the following program:

```
[object generator
  (state
   [filter := [create-filter <== [:new]]]
   [n := 1])
  (script
   (=> [:generate]
       [alarm-clock <= [:tick 50]]
       (loop
        [filter <= [:check (incf n)]])))]

[object alarm-clock
  (state [time := 1])
  (script
   (=> [:tick limit]
       (do () ((< limit time))
         [time := (1+ time)])
       (break-point "Suspended in the alarm clock.")
       [time := 0]))]
```

a break-point form with a string argument occurs in the definition of the object `alarm-clock`. The generator in this program sends a message to the alarm clock before generating integers. The alarm clock in turn repeats to execute an assignment form 50 times and then executes the break-point form.

```
---------------------------------------
<ABCL/1> [generator <= [:generate]]
```

```
2
3
5
7
11
13
17
Break: Suspended in the alarm clock.
<INSPECTING #<alarm-clock 0>> □
-------------------------------------
```

14.4 Stepwise Execution

In the inspector, the commands `:step` and `:stepme` are available for the purpose of stepwise execution.

The inspector command `:step` temporarily terminates the inspector and executes in a single step the active object which is currently at the head of the scheduling queue. Then this command re-invokes the inspector on this executed object. In contrast, the inspector command `:stepme` re-invokes the inspector just after the inspected object becomes at the head of the scheduling queue again and is executed in a single step. The command `:step` supports stepwise execution of the whole ABCL/1 system, whereas the command `:stepme` supports stepwise execution of a single object. Note that other objects may be executed while an object is executed in a single step.

At the top level, a *step-objects* form in the following syntax:

`(step-objects` *object* ··· *object*`)`

is available. After evaluation of this form, execution of any one of the specified objects invokes the inspector on the object. Notice that another step-objects form with no arguments:

`(step-objects)`

cancels the effects of the previous step-objects form.

```
-------------------------------------
<INSPECTING #<alarm-clock 0>> :ins generator

<INSPECTING #<generator 0>> :vs        ;; Listing the values of the state
state    filter   = #<filter 0>        ;; variables.
state    n        = 27        ;; Currently, the value of the state variable n is 27.

<INSPECTING #<generator 0>> :stepme    ;; Execution in a single step.
```

```
<INSPECTING #<generator 0>> :stepme    ;; Execution in a single step.

<INSPECTING #<generator 0>> :vs
state    filter   = #<filter 0>
state    n        = 28        ;; Now the value of the state variable n becomes 28.

<INSPECTING #<generator 0>> (show-system-queue)
*** ACTIVE objects  ***         ;; Listing the active and waiting objects.
#<filter 7>
#<filter 6>
#<filter 5>
#<filter 1>
#<filter 0>
#<generator 0>

*** WAIT-FOR objects ***
#<filter 4>
#<filter 3>
#<filter 2>

*** VALUE-WAIT objects ***


nil
<INSPECTING #<generator 0>> (step-objects (find-object 'filter 2))
   ;; The value of this find-object form is the object whose print-name is #<filter 2>
(#<filter 2>)     ;; and which is specified as the argument of the step-objects form.
<INSPECTING #<generator 0>> :c    ;; Continuing the suspended execution.

19
<INSPECTING #<filter 2>> □
---------------------------------------
```

In the above session example, we use a *find-object* form whose syntax is as follows:

(find-object *name number*)

The value of the find-object form is the object whose *print-name* is #<*name number*>[27].

[27] *name* can be an uninterned symbol.

15 Other Top Level Forms

The ABCL/1 system supports the following miscellaneous forms:

`(pwd)`
: This form prints the name of the current working directory.

`(cd` *directory-name*`)`
: This form changes the working directory to the *directory-name*.

`(disable-debug)`
: This form improves the execution efficiency with sacrifice of some debugging mechanisms. For instance, after this form is evaluated, interruptions from the keyboard are disabled and a show-system-queue form does not print any information about the objects in the waiting modes. Just after the ABCL/1 system is invoked, these debugging facilities are enabled.

`(enable-debug)`
: This form enables the debugging facilities which are disabled by a disable-debug form.

`(improve-room)`
: This form makes the storage size of the ABCL/1 system larger. The user should not type this form in the ABCL/1 system running on a machine with small (*e.g.*, 4M bytes) main memory.

`(abcl::gbc-message-on)`
: After evaluation of this form, the garbage collector works verbosely.

`(abcl::gbc-message-off)`
: After evaluation of this form, the garbage collector works silently.

16 The Summary of the Top Level Forms

`(bye)` or `(by)`
: halts the ABCL/1 system.

`?`
: prints the summary of the top level forms.

`(inc` *filename* ⋯`)`
: loads the specified ABCL/1 source, internal code, and compiled code files.

`(comp` *filename* ⋯`)`
: compiles the specified ABCL/1 source files.

`(show-objects)`
: prints all the objects that are created at the top level.

`(trace-objects` *object* ⋯`)`
: turns on the trace flags of the specified objects and returns the list of the currently traced objects or the symbol `:trace-all-objects`. If no arguments are supplied, it turns on the trace flags of all the objects.

`(untrace-objects` *object* ⋯`)`
: turns off the trace flags of the specified objects and returns the list of the currently traced objects. If no arguments are supplied, it turns off the trace flags of all the objects.

`(show-system-queue)`
: prints all the objects in the active and waiting modes.

`(reset` *object* ⋯`)`
: forces the specified objects to enter the dormant mode.

`(full-reset` *object* ⋯`)`
: forces the specified objects to enter the uninitialized mode. If no arguments are supplied, it forces all the objects that are created at the top level to enter the uninitialized mode.

`(history-on)`
: starts recording the event histories of all the objects.

`(history-off)`
: stops recording the event histories of all the objects.

`(history` *object* `&key :start :end)` prints the event history of the specified object in the interval specified by the key word parameters. The default value of the keyword parameters are 0 and $+\infty$.

`(inspect` *object*`)`
invokes the inspector to inspect the specified object.

`(describe` *object*`)`
prints the mode, message protocols, and values of state/environment variables of the specified object.

`(protocol` *object*`)`
prints the message protocols of the specified object.

`(cd` *directory-name*`)`
changes the working directory to the specified one.

`(pwd)`
prints the name of the current working directory.

`(step-objects` *object* $\cdots$`)`
makes each of the specified objects invoke the inspector when it is executed in a single step.

`(find-object` *object-name number*`)`
returns the object whose printing form is `#<`*object-name number*`>`.

`(where` *object*`)`
prints the contents of the program counter of the specified object.

`(disable-debug)`
improves execution efficiency with sacrifice of debugging facilities.

`(enable-debug)`
cancels the effects of a disable-debug form.

`(improve-room)`
makes the storage size of the ABCL/1 system larger. It is often useful to cope with frequent garbage collections.

`(abcl::gbc-message-on)`
lets the garbage collector work verbosely.

`(abcl::gbc-message-off)`
lets the garbage collector work silently.

17 The Summary of the Inspector Commands

`?`
: prints the inspector command summary.

`:man` *command-name*
: prints information about the specified command.

`:stat`
: prints the mode and the values of the state/environment variables of the inspected object.

`:where`
: prints the contents of the program counter of the inspected object.

`:proto`
: prints the message protocols of the inspected object.

`:mode`
: prints the current mode of the inspected object.

`:v,:vs,:ve,:vl`
: print the values of all the variables, the state variables, the environment variables, and the local (*i.e.*, temporary and pattern) variables, respectively, of the object being inspected.

`:l,:ls,:le,:ll`
: print the names of all the variables, the state variables, the environment variables, and the local (*i.e.*, temporary and pattern) variables, respectively, of the object being inspected.

`:ins` *form*
: evaluates the *form* under the current environment of the inspected object and recursively invokes the inspector on the evaluation result. This recursive level can be exited from by the inspector command `:ret`.

`:hist`
: prints the event history of the inspected object.

`:stepme`
: performs a stepwise execution of the inspected object. During execution, other active objects may also be executed. After the execution, the control returns to the inspector.

`:step`
: performs a stepwise execution of a single object, which is at the head of the scheduling queue. After that, the control returns to the inspector, which inspects the executed object.

`:ret`
: exits from the current recursive inspection level and moves up to the previous one.

`:c`
: halts the inspector and resumes the suspended computation (if it exists).

`:q`
: halts the inspector and forces all the active objects to enter the dormant mode.

PART III

18 Syntax of ABCL/1

In this section, we describe the syntax of the language ABCL/1 in terms of an extended BNF notion. We use the following two extended meta constructs:

- $\{\!\{A\}\!\}$ is equivalent to $\lambda|A|AA|AAA|\cdots$.
- $[\![A]\!]$ means that A is optional.

where λ is an empty string and A is some expression in our extended BNF.

```
<form> ::= <object-definition>
         | <message-send>
         | <!-notation>
         | <bracket-notation>
         | <parallel-send>
         | <wait-for>
         | <wait-for-loop>
         | <assignment-form>
         | <match>
         | <match-loop>
         | <lisp-form>

<object-definition> ::= [object [[ <id> ]]
                            [[ (state {{ <initialize-form> }} ) ]]
                              (script {{ <script-declaration> }} )
                            [[ (routine {{ <routine-definition> }} ) ]] ]

<id> ::= <lisp-form>
           where (and (symbolp '<id>) (not (constantp '<id>)))

<initialize-form> ::= <id> | [ <id> := <form> ]

<script-declaration> ::= ( <script-arrow> <message-pattern>
                            [[ @ <id>]] [[ from <id>]]
                            [[ where <lisp-form>]]
                            [[ (temporary {{ <initialize-form> }} ) ]]
                            {{ <form> }} )
```

```
<script-arrow> ::= => | =>>

<message-pattern> ::= <pattern>
                    | [ <keyword> {{ <id> }}
                            [[ & {{ <id> }} ]] [[ . <id> ]] ]

<keyword> ::= <lisp-form> where (keywordp '<keyword>)

<pattern> ::= <constant>
            | <id>
            | [ {{ <pattern> }} ]
            | [ <pattern> {{ <pattern> }} . <pattern> ]

<constant> ::= <lisp-form> where (constantp '<constant>)

<routine-definition> ::= (<id> <lambda-list> {{ <form> }} )

<lambda-list> ::= Defined by [?]

<message-send> ::= <past-send> | <now-send> | <future-send>
                 | <exp-past-send> | <exp-now-send> | <exp-future-send>

<past-send> ::= [ <form> <= <form> [[ @ <form> ]] ]

<now-send> ::= [ <from> <== <form> ]

<future-send> ::= [ <form> <= <form> $ <form> ]

<exp-past-send> ::= [ <form> <<= <form> [[ @ <form> ]] ]

<exp-now-send> ::= [ <from> <<== <form> ]

<exp-future-send> ::= [ <form> <<= <form> $ <form> ]

<!-notaion> ::= !<form>

<bracket-notation> ::= []
                     | [ <bracket-element-1> ]
                     | [ <bracket-element-1> <bracket-element-2>
                             {{ <form> }} ]
```

```
                    | [ <form> {{ <form> }} .  <form> ]

<bracket-element-1> ::= <form>
                            where (not (eq '<bracket-element-1> 'object))

<bracket-element-2> ::= <form>
                            where (not (member '<bracket-element-2>
                                               '(:= <= <<= <== <<==)))

<parallel-send> ::= { {{ <message-send> }} }

<wait-for> ::= (wait-for {{ <script-declaration> }} )

<wait-for-loop> ::= (wait-for-loop {{ <script-declaration> }} )

<assignment-form> ::= [<id> := <form>]

<match> ::= (match <form>
                    {{ (is <pattern> [[ where <form> ]] {{ <form> }} ) }}
                    [[ (otherwise {{ <form> }} ) ]] )

<match-loop> ::= (match-loop <form>
                    {{ (is <pattern> [[ where <form> ]] {{ <form> }} ) }}
                    [[ (otherwise {{ <form> }} ) ]] )

<lisp-form> ::= Defined by [?]
```

19 The ABCL/1 Mode in GNU Emacs

The distribution tape of the ABCL/1 system includes the Emacs Lisp file `abcl1.el`, which contains support tools for development of ABCL/1 programs in GNU Emacs. In order to use these support tools, first move `abcl.el` to an appropriate directory which is in the *load paths* of GNU Emacs and then put the following lines:

```
------------------------------------------------------------------------
(global-set-key "\M-A" 'run-abcl1)
(setq auto-mode-alist
      (cons '("\\.abcl1$" . abcl1-mode) auto-mode-alist))
(autoload 'abcl1-mode "abcl1")
(autoload 'run-abcl1 "abcl1")
------------------------------------------------------------------------
```

in the file `~/.emacs` of each user.

Currently, `abcl1.el` includes the following commands:

`abcl1-mode`
: This command changes the major mode of the current buffer to *abcl1-mode*.

`run-abcl1` (M-A)
: This command invokes the ABCL/1 system as an inferior process of GNU Emacs. Input and output operations are performed via the buffer `*abcl1*`. If the ABCL/1 system has already been running in the buffer the only effect is that the current buffer becomes `*abcl1*`.

`abcl1-send-file-and-go` (M-X)
: This command saves the contents of the current buffer in the associated file and then lets the ABCL/1 system in `*abcl1*` load the file. If the ABCL/1 system has not yet been running, this command invokes the ABCL/1 system, too.

`abcl1-documentation` (C-H A)
: This command interactively asks an argument. If the argument supplied by the user is a function name, this command prints its *document string*. If the argument is an ABCL/1 reserved name such as `object`, this command prints the syntax of forms beginning with the name.

`abcl1-next-error` (M-‘)
: This command reads an expression:

    `@`(*file-name number number* ···)

printed by a *where* form or a *:where* command and searches the location represented by the expression. Precisely, this command displays the file *file-name* and puts the cursor at the specified location. If more than one locations are printed, the cursor goes to each position by repeated invocations of this command.

20 Known Bugs and Features

1. The current implementation of the ABCL/1 system is running on a single processor machine and realizes pseudo parallel execution using a scheduling queue and a context switching mechanism. However, context switching cannot occur within the following portions:

   (a) lambda expressions

   (b) constraints following message patterns

   Therefore, the following forms:

   (a) message passing forms

   (b) tagbody special forms containing tags

   (c) block special forms

   (d) wait-for and wait-for-loop forms

   (e) routine calls

   (f) atomic, non-resume, and suicide forms

   which may cause context switching can appear neither in *lambda expressions* nor *constraints*.

   Notice that *loop*, *do*, and *do** macro calls of Common Lisp are expanded to *tagbody* forms. Thus, in the current implementation of ABCL/1, any lambda expression cannot include such loop constructs.

2. The current implementation does not allow the user to use *return-from* and *go* special forms for exiting from wait-for(-loop) forms.

3. Let and let* forms cannot bind special variables.

4. The current implementation ignores declare special forms.

5. Though the contents of any environment or pattern variable cannot be updated according to the language specification (see Section 7), the current implementation allows an object to update the contents of its environment and pattern variables.

6. When a lambda closure created by an object is transmitted to another object, the system does not assure what will happen.

7. In the inspector, any form which may cause context switching cannot be evaluated. Moreover, neither let nor let* special forms can be evaluated in the inspector.

8. When a source program includes macro calls, *where* forms and *:where* commands may be erroneous.

9. Every object is displayed by the system in the form of:

   #<*object-name number***>**

   However, under the current implementation, two objects defined by different object definition forms with the same object name may share the same printing form. Therefore, a find-object form may not return an expected result.

10. Protocol forms may not print enough information.

21 Caution!!!

The distribution tape of the ABCL/1 system includes a copy of KCl (Kyoto Common Lisp). Therefore, we cannot distribute the ABCL/1 system to any site without the KCl license. Those who would like to get the license of KCl should contact:

siglisp%kurims.kurims.kyoto-u.junet%utokyo-relay@relay.cs.net

Thank you.