A CLOS protocol for lexical environments

Robert Strandh
robert.strandh@gmail.com
Unaffiliated

Irène Durand
irene.durand@u-bordeaux.fr
LaBRI, University of Bordeaux
Talence, France

ABSTRACT

The concept of an *environment* is mentioned in many places in the Common Lisp standard, but the nature of the object is not specified. For the purpose of this paper, an environment is a mapping (or several such mappings when there is more than one namespace as is the case for Common Lisp) from *names* to *meanings*.

In this paper, we propose a replacement for the environment protocol documented in the book "Common Lisp the Language, second edition" by Guy Steele. Rather than returning multiple values as the functions in that protocol do, the protocol suggested in this paper is designed so that functions return instances of standard classes. Accessor functions on those instances supply the information needed by a compiler or any other *code walker* application.

The advantage of our approach is that a protocol based on generic functions and standard classes is easier to extend in backward-compatible ways than the previous protocol, so that implementations can define additional functionality on these objects. Furthermore, CLOS features such as auxiliary methods can be used on these objects, making it possible to extend or override functionality provided by the protocol, for implementation-specific purposes.

CCS CONCEPTS

• **Software and its engineering → Compilers**;

KEYWORDS

CLOS, Common Lisp, Environment, Compilation

ACM Reference Format:
Robert Strandh and Irène Durand . 2022. A CLOS protocol for lexical environments. In *Proceedings of the 15th European Lisp Symposium (ELS'22).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.5281/zenodo.6331519

1 INTRODUCTION

The Common Lisp standard [1] contains many references to environments. Most of these references concern *lexical* environments at *compile time*, because they are needed in order to process forms in non-null lexical environments. The standard does not specify the nature of these objects, though in the book "Common Lisp, the Language, second edition" [4] (henceforth referred to as "CLtL2") there is a suggested protocol that is supplied by some existing Common Lisp implementations.

The protocol documented in CLtL2 has several problems. Functions in the protocol return multiple values, a fact that makes the protocol hard to extend. Furthermore, the protocol is incomplete. A typical compiler needs more information than the protocol provides, making implementation-specific extensions obligatory for the protocol to be useful. For that reason, although existing Common Lisp implementations often provide such extensions, the CLtL2 protocol is not what the native compiler of the implementation actually uses.

In this paper, we propose a modern alternative protocol based on CLOS. Rather than returning multiple values, our protocol functions return instances of standard classes. Accessors for those instances can be used by compilers and other *code walker* applications in order to obtain the information needed for the task to be accomplished. This protocol is defined and implemented in the Trucler library.[1]

Two of the functions in the section about environments in CLtL2 are not discussed in this paper, namely `parse-macro` and `enclose`. These functions do not contribute any functionality to the protocol being described, and the interface provided by these functions does not require any modifications, which is why they are not discussed here. The function `enclose` requires an evaluator such as a compiler or an interpreter, and the evaluator will certainly *use* the functionality in the protocol, but not add to it. The function `parse-macro` does not seem to even use this functionality, and indeed the optional *env* parameter of this function is declared `ignore` both in SBCL and CCL.

Although `parse-macro` and `enclose` are essential for any code-walking application, the purpose of the current work is not to provide a complete implementation-independent library for code walking, but just to propose an alternative protocol for accessing lexical environments.

2 PREVIOUS WORK

In this section, we describe how different implementations of Common Lisp represent lexical environments, and whether these implementations include a version of the protocol described in CLtL2. For commercial implementations, we include only their documented version of the CLtL2 protocol. We start by presenting the details of the CLtL2 protocol as described in the book.

[1] https://github.com/s-expressionists/Trucler

ELS'22, March 21–22 2022, Porto, Portugal

https://doi.org/10.5281/zenodo.6331519

2.1 Common Lisp the Language, second edition

Section 8.5 of CLtL2 describes a set of functions for obtaining information from an environment object, for creating a new such object by augmenting an existing one, and two more operators related to environments that are outside the scope of this paper, i.e., `parse-macro` and `enclose`.

In this section, we provide an overview of that protocol, and we give an assessment of its usefulness in the context of a language processor.

2.1.1 Environment query. For environment query, the protocol defines three functions. We describe them briefly here.

The function `variable-information` takes a symbol and an optional environment object as arguments. It returns three values. The first value indicates the type of the binding (lexical variable, special variable, symbol macro, constant variable) or `nil` if there is no binding or definition in the environment for that symbol The second value is a Boolean, indicating whether the binding is local or global. The third value is an association list containing declarations that apply to the binding.

The function `function-information` takes a function name and an optional environment as arguments. Again, three values are returned. The first value indicates the type of the binding (function, macro, special operator[2]) or `nil` if there is no binding or definition in the environment for that function name. As before, the second value indicates whether the definition is local or global, and the third value is an association list of declarations that apply.

The function `declaration-information` is used in order to query the environment for declarations that do not apply to any particular binding in the environment. It takes a *declaration identifier*[3] and an optional environment as arguments. The declaration identifier can be the symbol `optimize`, the symbol `declaration`, or some implementation-defined declaration identifier. It returns a single value that contains information related to the corresponding declaration identifier.

To begin with, it is clear that this set of functions is insufficient to process all Common Lisp code, because no mechanism is described for querying the environment for information related to *blocks* and *go tags*. Functions for this purpose are provided as extensions by Allegro Common Lisp as described in Section 2.7, and by LispWorks as described in Section 2.8.

2.1.2 Environment augmentation. For augmenting an environment, i.e., creating a new, augmented, environment from an existing one, the same section describes the function `augment-environment`. This function has a keyword parameter for each type of object to be added to the current lexical environment: `:variable`, `:symbol-macro`, `:function`, `:macro`, and `:declare`. Furthermore, each argument is a list of lexical definitions, thereby allowing an arbitrary number of mappings to be added to an environment in order to create an augmented environment.

2.1.3 Assessment of the protocol. In general, the protocol as described in the book is insufficient for use in any but the simplest kind of language processor. Even if query functions are added for tags and blocks, and additional keyword argument are added to the function `augment-environment` for tags and blocks, we argue that the protocol would still be insufficient.

Any non-trivial language processor would need for a function such as `function-information` to return information about the function, other than related declarations. At the very least, information such as the lambda list of the function, and information needed for inlining, would have to be included.

The protocol could obviously be extended to allow for such information, but such extensions would involve incompatible additions such as more return values. Furthermore, none of the Common Lisp implementations we investigated use this protocol internally, which is an indication that the compiler needs more information than the protocol provides. And none of the implementations we investigated provide extensions that would allow the use of the protocol in a non-trivial language processor.

2.2 SBCL

2.2.1 Native. SBCL[4] defines a structure class `lexenv`. Instances of this class are passed as the `&environment` argument to macro expanders and other functions that take lexical environment objects as arguments.

This structure class contains several slots, and in particular:

- An association list of information about defined functions. The name of the function is used as a key.
- An association list of information about defined variables. The name of the variable is used as a key.
- An association list of information about blocks, The name of the block is used as a key.
- An association list of information about `tagbody` tags. The name of the tag is used as a key.

2.2.2 CLtL2. The distribution of SBCL contains a contribution that supplies some of the funtionality described in the book CLtL2 but that was not included in the Common Lisp standard. Part of this contribution is an implementation of the environment protocol of CLtL2.

2.3 CCL

2.3.1 Native. CCL[5] defines a class `lexical-environment` which is a special kind of class called an `istruct`. Classes of this type are represented as lists of slots rather than as

[2]The term used in the book is *special form*, but the terminology has been improved since then

[3]The term used in the book is *name* and the parameter is called *decl-name*, but the terminology has changed since then.

[4]http://ww.sbcl.org/

[5]https://ccl.clozure.com/

standard objects as would normally be the case, probably for reasons of bootstrapping.

2.3.2 CLtL2. CCL has implementations of the functions defined in CLtL2. These functions take a native lexical environment as an optional argument.

2.4 CMUCL

2.4.1 Native. A lexical environment is an instance of the structure class `lexenv`. There is a slot for each type of entry, i.e., `functions`, `variables`, `blocks`, `tags`, and some other slots for implementation-specific details. Each of the main slots contains an association list in which the name is the key and the value contains associated information for the name.

Access to the lexical environment is provided by the macro `lexenv-find` and the function `lexenv-find-function`. These operators do not take an environment object as an argument, and instead access this object as the value of the special variable `*lexical-environment*`. And `lexenv-find-function` is a wrapper for a call to `lexenv-find` with a specific `:test` function for the key of the association list containing functions.

2.4.2 CLtL2. CMUCL provides definitions of the functions defined in CLtL2. The code for these functions is defined in the package `ext`. No extensions are provided for tags or blocks.

2.5 ECL

2.5.1 Native. The native compilation environment of ECL[6] is represented as a single `cons` cell where the `car` is a list of *variable records* and the `cdr` is a list of *macro records*. Information about blocks and tags is included in the list of *variable records*. With few exceptions, a record is a list with the name of the entity in the `car`. Records for blocks and tags are distinguished by having a keyword symbol `:block` or `:tag` in the `car` of the list representing the record.

2.5.2 CLtL2. Currently, ECL does not offer a CLtL2-compatible interface to its lexical environments. Some work has been done to create such an interface, but it is still work in progress.

2.6 Clasp

2.6.1 Native. The native compilation environment of Clasp[7] is currently that used in early versions of the Cleavir[8] compiler framework. Ultimately, Clasp will use Trucler as described in Section 3.

2.6.2 CLtL2. Clasp provides an implementation of the CLtL2 protocol. The code is present in the package named `clasp-cltl2`. The function `augment-environment` has two additional keyword arguments, namely `tag` and `block`. However, no extension allows for client code to access information about blocks and tags.

[6] https://common-lisp.net/project/ecl/
[7] https://github.com/clasp-developers/clasp
[8] https://github.com/s-expressionists/Cleavir

2.7 Allegro

2.7.1 Support for CLtL2 protocol. The documentation for Allegro Common Lisp contains a separate document describing their protocol for environments in the spirit of CLtL2.[9] We summarize the differences between the Allegro implementation and the CLtL2 protocol here.

- Information about blocks and tags have been added in the form of two new functions `block-information` and `tag-information`.
- The function `augment-environment` accepts additional keywords arguments such as `:block`, `:tag`, etc. in order to make it possible to augment an environment with all relevant information that the language processor may encounter.
- The function `augment-environment` accepts an additional keyword argument `:locative` that can be used by client code to supply additional information about the entity, for example the value of a constant variable. The query functions return an additional value which is the information supplied to `augment-environment`.
- The order and the number of the return values of the query functions have been modified, so as to allow for the additional *locative* value, and to have frequently used return values before the less frequently used.
- Several other features have been added to the protocol in order to make it a complete tool for a language processor, and for the purpose of minimizing memory allocation. These additional features are outside the scope of this paper.

2.8 LispWorks

2.8.1 Support for CLtL2 protocol. The documentation for LispWorks Common Lisp describes the operators that implement the CLtL2 protocol. These operators are available in the `hcl` package.

Like Allegro, LispWorks also provides the functionality for blocks and tags that is missing from the CLtL2 protocol, but instead of adding functions `block-information` and `tag-information`, LispWorks provides a single additional function named `map-environment`. This function has a single required parameter, namely an environment object. It has four keyword parameters: *variable*, *function*, *block*, and *tag*. Each corresponding argument is a designator for a function that can accept three arguments: *name*, *kind*, and *info* as follows:

- The function *variable* is called for each local variable binding in the environment. *name* is the name of the variable, *kind* is one of `:special`, `:symbol-macro` or `:lexical`, with the same meaning as for the function `variable-information` in the CLtL2 protocol. When *kind* is `:symbol-macro`, then *info* is the expansion; otherwise, *info* is unspecified.
- The function *function* is called for each local function in the environment. *name* is the name of the function,

[9] https://franz.com/support/documentation/current/doc/environments.htm

kind is one of `:macro` or `:function`, with the same meaning as for the function `function-information` in the CLtL2 protocol. When *kind* is `:macro`, then *info* is the macro-expansion function; otherwise, *info* is unspecified.
- The function *block* is called for each block in the environment. *name* is the name of the block, *kind* is the keyword symbol `:block`, and *info* is unspecified.
- The function *tag* is called for each tag in the environment. *name* is the name of the tag, *kind* is the keyword symbol `:tag`, and *info* is unspecified.

However, when `map-environment` calls the function in the keyword argument *function* and the name of the function is of the form `(setf` *symbol*`)`, then the argument is not the name of the function, but instead a symbol that is used internally in LispWorks to name the function. According to the maintainer of LispWorks, this restriction will be removed in future versions.

Similarly, the keyword argument `:function` of the function `augment-environment` must be a list of symbols. To represent a function with a name of the form `(setf` *symbol*`)`, the internal symbol used by LispWorks must be passed, rather than the true name of the function. Again, this restriction will be removed in future versions.

2.9 CLtL2 compatibility system

The system `cl-environments`[10] provides a compatibility layer that allows client code to use the CLtL2 environment protocol independently of the Common Lisp implementation. Supported Common Lisp implementations are CLISP, CCL, ECL, ABCL, CMUCL, SBCL, Allegro, and LispWorks.

This library does not provide additional operators for querying the environment for tags or blocks, nor does it provide keyword arguments on `augment-environment` for augmenting an environment with such information.

2.10 Software including a code walker

In his paper presented at the European Lisp Symposium 2017 [3], Raskin gives an overview of various libraries that require code walking. In that paper, he also argues that it is impossible to write a completely portable code walker, although he addresses many of the difficulties in his own, mostly portable, code walker named Agnostic Lizard.

In particular, one of the libraries he mentions in his paper is hu.dwim.walker. This library provides a general-purpose configurable code walker. It uses its own protocol for accessing and augmenting the environment. This protocol resembles the one presented in this paper in some ways.

3 OUR TECHNIQUE

We define a CLOS-based protocol for accessing and augmenting a lexical environment. This protocol is defined and implemented in the Trucler library.

[10] https://github.com/alex-gutev/cl-environments

3.1 Querying the environment

A language processor calls one of the query functions in order to determine the nature of a language element, depending on the position in source code of that language element. All these functions are generic, and they all take a `client` parameter and an `environment` parameter. Methods defined by Trucler do not specialize to the `client` parameter. Client code should pass an object specific to the application as a value of that parameter, and it can supply methods specialized to the class of this object, for the purpose of extending or overriding default behavior. The `environment` parameter is an object of the type used by the implementation that Trucler is configured for. Functions that are used to query a particular *name* have an additional parameter for this purpose.

The following query functions are defined by Trucler. Each one returns an instance of a class that allows the language processor to determine the exact nature of the language element (`nil` is returned if there is no definition for the element), for example by using the instance in a call to a generic function:

- `describe-variable`. This function returns an instance of a class that distinguishes lexical variables, special variables, constant variables, and symbol macros.
- `describe-function`. This function returns an instance of a class that distinguishes global functions, local functions, and macros.
- `describe-block`.
- `describe-tag`.
- `describe-optimize`.
- `describe-declarations`. This function is called by the language processor in order to determine the declaration identifiers of `declaration` proclamations.

3.2 Augmenting the environment

A language processor calls one of the augmentation functions in order to define a lexical environment within the scope of a declaration or a definition encountered in source code. All these functions take at least a `client` parameter and an `environment` parameter just like the query functions, and they all return a new lexical environment, augmented according to the function being called.

The following functions are called by the language processor when a local definition is encountered, and they return a new environment that includes the new definition:

- `add-lexical-variable`.
- `add-special-variable`.
- `add-local-symbol-macro`.
- `add-local-function`.
- `add-local-macro`.
- `add-block`.
- `add-tag`.

The following functions are called by the language processor as the result of a local declaration that restricts an existing local function or variable:

- `add-variable-type`.

- `add-variable-ignore`.
- `add-variable-dynamic-extent`.
- `add-function-type`.
- `add-function-ignore`.
- `add-function-dynamic-extent`.

The following functions are called by the language processor as the result of a local `optimize` declaration.

- `add-inline`.
- `add-speed`.
- `add-compilation-speed`.
- `add-debug`.
- `add-safety`.
- `add-space`.

3.3 Restricting the environment

Recall that the description of the function `enclose` in section 8.5 of CLtL2 mentions that the consequences are undefined if the *lambda-expression* argument contains references to entities in the environment that are not available at compile time, such as lexically visible bindings of variable and functions, `go` tags, or block names.

As a service to a robust implementation of the `enclose` function, the Trucler library provides a function named `restrict-for-macrolet-expander` that takes an environment as an argument and returns an environment that contains only entities available at compile time. Using this function, the implementation of `enclose` can return a function that will signal an error if the *lambda-expression* argument contains unavailable references.

3.4 The reference implementation

Trucler supports some existing Common Lisp implementations as described in Section 3.5, but it also comes with a *reference implementation* that can be used by a new Common Lisp implementation that does not have its own representation of lexical environments. The reference implementation is used by SICL[11] for instance.

In the reference implementation, a lexical environment is represented as a standard object containing a slot for each type of description to be returned by a query function as described in Section 3.1. Each slot contains a list of descriptions ordered from innermost to outermost. A query function merely returns the first item on the list that matches the name that was passed as an argument to the query function. As a direct consequence of this representation, there is no performance penalty in the query functions, due to the fact that a new environment is created for every call to an augmentation function.

In order to create new objects such as environments or descriptions, we use a technique that we call *quasi cloning*. A generic function named `cloning-information` is called with the original object as an argument. This function then returns a list of pairs. The first element of the pair is a slot initialization argument for the class of the object and the second element of the pair is the name of a slot reader for the same slot. This information is then used to access slots in the original object and to pass that information as an initialization argument to `make-instance`. We call it quasi cloning, because some new value is prepended to the initialization arguments so that the copy is like the original, except for one slot.

The advantage of quasi cloning is that Trucler does not need to know the right class to instantiate. It creates an instance of the same class as the original object, and that class can be defined by client code. Client code must define a method on `cloning-information`, but this generic function uses the `append` method combination, so that only slots defined by client code need to be mentioned in that method.

Occasionally, an entirely new instance of some class must be created, rather than being obtained by quasi cloning an existing instance. This situation occurs when information about a new item such as a local variable or a local function must be used to augment an existing environment. To allow Trucler to create an instance of a class that has been determined by client code, Trucler first calls what we call a *factory* function. This function takes the `client` object and returns the class metaobject to instantiate. For example, to create an instance of a class that describes lexical variables, Trucler calls the function `lexical-variable-description-class`, passing it the `client` object supplied by client code. The default method on this generic function returns the default class used by the reference implementation, but client code that needs additional information about lexical variables may create a subclass of the default class, and a method on `lexical-variable-description-class` that returns this subclass.

3.5 Supported Common Lisp implementations

Trucler currently provides support for SBCL and CCL. Contributions for other Common Lisp implementations are welcome. With these implementations, it is possible to write code walkers that are portable across different Common Lisp implementations. In particular, a Cleavir-based compiler can compile source code for any of the supported implementations.

3.6 Examples

In this section, we show some examples of how Trucler can be used by a code walker. All examples are from Cleavir used in the SICL compiler. We have simplified the examples compared to the actual code, in order to avoid too much clutter. For example, we have omitted the handling of error situations and restarts.

The part of Cleavir that uses Trucler is the phase that converts a *concrete syntax tree* (CST) to an *abstract syntax tree* (AST). A concrete syntax tree can be thought of as a Common Lisp expression but where each sub-expression is wrapped in a standard object that holds additional information such as source location. At the core of this compilation

[11] https://github.com/robert-strandh/SICL

phase is the generic function `convert-cst`. For each class of description objects that Trucler can return, this generic function has a method specialized to that class.

The function `convert-cst` is called by a top-level function `convert` that determines the structure of the expression to convert and calls the appropriate Trucler query function and then invokes `convert-cst` with the object returned by Trucler.

The method specialized to `local-macro-description` looks like this:

```
(defmethod convert-cst
    (client
     cst
     (info trucler:local-macro-description)
     environment)
  (let* ((expander (trucler:expander info))
         (expanded-form
          (expand-macro expander cst environment))
         (expanded-cst
          (cst:reconstruct expanded-form cst client)))
    (setf (cst:source expanded-cst) (cst:source cst))
    (with-preserved-toplevel-ness
      (convert client expanded-cst environment))))
```

As we can see, this method is specialized to the Trucler class `local-macro-description`, and no other parameter is specialized. The code calls the accessor `expander` on the `info` parameter, which returns the macro expander associated with the local macro.

The function `expand-macro` is responsible for taking into account `*macroexpand-hook*` as the Common Lisp standard requires. The call to `reconstruct` has to do with preserving source information in the expanded form. The essence of the body is the call to `convert` which converts the expanded form (wrapped in a concrete syntax tree).

The next example shows how the environment is augmented when a `block` special form is converted:

```
(defmethod convert-special
    (client
     (symbol (eql 'block))
     cst
     environment)
  (cst:db origin (block-cst name-cst . body-cst) cst
    (declare (ignore block-cst))
    (let ((name (cst:raw name-cst)))
      (let* ((ast (cleavir-ast:make-ast
                   'cleavir-ast:block-ast))
             (new-environment
               (trucler:add-block
                client environment name ast)))
        (setf (cleavir-ast:body-ast ast)
              (process-progn
               client
               (convert-sequence
                client body-cst new-environment)
               environment))
        ast))))
```

In the example above, `cst:db` is a version of the standard Common Lisp operator `destructuring-bind`, that is used to destructure concrete syntax trees as opposed to ordinary Common Lisp source expressions. The last argument to `add-block` is an optional argument that Trucler calls `identity` and that Trucler stores in the environment, associated with the block information. The nature of the object supplied is entirely determined by client code. In our case, we supply an abstract syntax tree that represents the `block` special form so that when an associated `return-from` is found, the two abstract syntax trees can be connected.

The essence of the method body is the call to the function named `convert-sequence` which converts the body of the `block` form in the original environment augmented with information about the `block` form.

4 BENEFITS OF OUR TECHNIQUE

The query functions of our protocol are generic functions, allowing client code to define methods for overriding or extending default behavior. For this purpose, the query functions all have a `client` parameter. Default methods supplied by Trucler do not specialize to this parameter, but client code should supply a standard object as the corresponding argument when these functions are called. The class of this argument can then be used in primary or auxiliary methods defined by client code, thereby allowing arbitrary customization of the library.

Furthermore, each query function returns an instance of a standard class, rather than multiple values. Client code can define subclasses of the classes used by the query functions. In particular, for objects in the global environment, client code can return instances of classes containing arbitrary information that it finds useful for the language processor. For example, if a global function turns out to be a generic function, client code can then return a subclass of the Trucler class `global-function-description` that contains information such as the the generic-function class, the method class, and the method combination, as we suggested in our paper about `make-method-lambda` [2].

5 DISADVANTAGES OF OUR TECHNIQUE

Compared to the protocol defined in CLtL2, our protocol probably involves more memory allocation, or "consing". Multiple values are likely to be handled without memory allocation in most high-end Common Lisp implementations, whereas our query functions return standard objects which obviously need to be allocated. Initializing the slots of these standard objects also comes with an additional cost.

To make things worse, our protocol is able to add a single mapping for each call to a protocol function, whereas the CLtL2 protocol function `augment-environment` is able to add an arbitrary number of mappings with a single call.

Our protocol consists of generic functions, and in implementations with a mediocre implementation of generic dispatch, our protocol can require more resources for function calls. Furthermore, the multiple values returned by the CLtL2 protocol are likely transmitted to the caller in registers or

some other relatively direct location, whereas the information returned by our query functions is present in slots of the standard objects being returned. Accessing this information involves calling a slot reader, which involves another call to a generic function.

However, we believe that the work done by the code walker of a compiler is small compared to that of other compilation phases such as optimization of intermediate code.

6 CONCLUSIONS AND FUTURE WORK

We have defined a CLOS-based protocol for lexical environments. This protocol can be used by any code walker such as a compiler or a version of `macroexpand-all`. Compared to the protocol defined in CLtL2 [4], ours is complete in that it has operators for querying an environment for references to tags and blocks, and for augmenting environments with such entities.

Furthermore, our protocol is implemented in the Trucler library. Trucler supplies implementations for some existing Common Lisp implementations, currently SBCL and CCL. The library also contains a *reference implementation* that can be used in new Common Lisp implementations that do not have an existing native representation of lexical environments, such as SICL and Clasp.

Future work involves adding more supported existing Common Lisp implementations. Individual implementations may require additional protocol functions, but such functions will have names in a package that is specific to the implementation.

Future work also involves investigating what new functionality might be required in the reference implementation in order to support specific requirements of new Common Lisp implementations that choose to use an extended version of the Trucler reference implementation.

7 ACKNOWLEDGMENTS

We would like to thank Jan Morningen and Karsten Poeck for reading an early draft of the paper and for suggesting improvements. Furthermore, we would like to thank Martin Simmons for explaining the wordings in the documentation of the CLtL2 protocol for LispWorks.

REFERENCES

[1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp*. American National Standards Institute, 1994.

[2] Irène Durand and Robert Strandh. MAKE-METHOD-LAMBDA revisited. In Nicolas Neuss, editor, *Proceedings of the 12th European Lisp Symposium (ELS 2019), Genova, Italy, April 1-2, 2019*, pages 20–23. ELSAA, 2019. doi: 10.5281/zenodo.2634303. URL https://doi.org/10.5281/zenodo.2634303.

[3] Michael Raskin. Writing a best-effort portable code walker in Common Lisp. In *Proceedings of the 10th European Lisp Symposium (ELS 2017), Brussels, Belgium, April 3-4, 2017*, pages 98 – 105. ELSAA, April 2017. doi: 10.5281/zenodo.3254669. URL https://doi.org/10.5281/zenodo.3254669.

[4] Guy L. Steele, Jr. *Common LISP: The Language (2Nd Ed.)*. Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.