CommonLisp$_0$ and CLOS$_0$: The Language Definition

Wolfgang Goerigk, Ulrich Hoffmann, Heinz Knutzen

Christian-Albrechts-Universität zu Kiel

Task: Definition effizienter Teilsprachen
Status: Draft
Document Identifier: APPLY/CAU/II.2/2
Created: 09.08.93
Revised: 03.09.93

A Practicable and Portable Lisp Ymplementation — CAU GMD ISST VW–GEDAS

Synopsis

This document includes the definition of the languages COMMONLISP$_0$ and CLOS$_0$ as strict subsets of COMMONLISP and CLOS.

This work was supported by the German Federal Ministry for Research and Technology (BMFT) within the joint project APPLY. The partners in this project are the Christian Albrechts University Kiel, the Fraunhofer Institute for Software Engineering and Systems Engineering (ISST), the German National Research Center for Computer Science (GMD), and VW-Gedas.

Christian Albrechts University Kiel
Institute for Computer Science and Practical Mathematics
Research Group "Programming Languages and Compiler Construction"
Preusserstr. 1-9, W–2300 Kiel 1, FRG
Fax: 0431/5661-43
Person to contact: Dr. Friedemann Simon
Phone: 0431/5604-25
E-mail: fs@informatik.uni-kiel.dbp.de

German National Research Center for Computer Science (GMD)
Institute for Applied Information Technology
Artifical Intelligence Research Division
P.O. Box 1316, W–5205 Sankt Augustin 1, FRG
Fax: 02241/14-2072
Person to contact: Prof. Dr. Thomas Christaller
Phone: 02241/14-2678
E-mail: thomas.christaller@gmd.de

Fraunhofer Institute for Software Engineering and Systems Engineering (ISST)
Kurstr. 33, O–1086 Berlin, FRG
Fax: 030/20372-301
Person to contact: Dr. Horst Friedrich
Phone: 030/20372-342
E-mail: horst.friedrich@isst.fhg.de

VW-GEDAS
Pascalstr. 11, W–1000 Berlin 10, FRG
Fax: 030/39007999
Person to contact: Dipl.-Ing. Jürgen Beste
Phone: 030/39007-331
E-mail: beste%sun1@hwsw.gedas.de

Contents

1 Introduction **1**

1.1 How to read this document 1

1.2 A strict subset of CommonLisp 2

1.3 Notation 3

2 CommonLisp$_0$ **3**

2.1 Introduction 4

2.2 Data Types 4

2.3 Scope and Extent 5

2.4 Type Specifiers 5

2.5 Program Structure 6

2.6 Predicates 7

2.7 Control Structure 7

2.8 Macros 8

2.9 Declarations 9

2.10 Symbols 9

2.11 Packages 9

2.12 Numbers 11

2.13 Characters 11

2.14 Sequences 11

2.15 Lists 12

2.16 Hash Tables 12

2.17 Arrays 12

2.18 Strings 12

2.19 Structures 12

2.20 The Evaluator 12

2.21 Streams 13

2.22 Input/Output 13

2.23 File System Interface 13

2.24 Errors 13

2.25 Miscellaneous Features 13

3 CL$_{macro}$ **13**

4 CLOS$_0$ **14**

4.1 Programmer Interface Concepts 15

4.1.1 Classes 16

4.1.2 Inheritance 17

4.1.3 Integrating Types and Classes 17

4.1.4 Determining the Class Precedence List 17

4.1.5 Generic Functions and Methods 17

4.1.6 Method Selection and Combination 17

4.1.7 Meta-objects 18

4.1.8 Object Creation and Initialization 18

4.1.9 Redefining Classes 18

4.1.10 Changing the Class of an Instance 19

4.1.11 Reinitializing an Instance 19

4.2 Functions in the Programmer Interface 19

5 Acknowledgements **21**

1 Introduction

The program development process for Lisp applications usually takes place within an interactive development and runtime system. It benefits from the possibilities to incrementally add and redefine arbitrary parts of the program and to interactively test and debug even incomplete program parts. Since many of the tools used to facilitate this process have been expressed in Lisp itself, it is easy to build an application specific development environment. Unfortunately, many of these tools actually have become part of the language CommonLisp itself and, as a consequence, the application specific program development environment often becomes part of the application program. Hence, many applications inherently include the ability to dynamically change their own program text, i.e. are *reflective*.

One of the major goals of the APPLY project is to develop the techniques needed for generating efficient stand alone applications which easily can be integrated with and into standard software environments. One of the techniques we use in order to achieve this goal is called *complete compilation* of programs or modules. It uses the complete set of information about the whole program or module in order to perform global and interprocedural program analysis and optimization.

In order to prove those optimizations correct we have to ensure that the program is not reflective, i.e. that it does not include the ability to change its own program text. In general, this is impossible for the full CommonLisp language. Hence, this document defines a suitable subset, essentially excluding the reflective parts of CommonLisp. We call this subset CommonLisp$_0$. It is a very large and strict subset of the CommonLisp as it is defined in [Steele Jr., 1984] and [Steele Jr., 1990].

1.1 How to read this document

This document has to be read carefully in conjunction with the original CommonLisp language definition, i.e. a copy of [Steele Jr., 1984] or [Steele Jr., 1990]. We will define CommonLisp$_0$ by stating the differences and changes with respect to the CommonLisp definition instead of listing the whole language. The second edition of Guy Steele's book ([Steele Jr., 1990]) contains a complete copy of the first one, and additionally contains

- corrections to the first edition which concern errors and inconsistencies,
- clarifications formulated by the ANSI standardization committee X3J13,
- extensions proposed by X3J13, and
- changes proposed by X3J13.

The ANSI standardization committee X3J13 has been established in order to define an ANSI standard for the language CommonLisp. Up to now, this standardization process is not yet finished. Therefore, the proposals of X3J13 stated in [Steele Jr., 1990] have to be taken as part of an ongoing discussion about the language, which at the end might result in a language definition for ANSI CommonLisp even is incompatible with [Steele Jr., 1990].

Therefore, at present the COMMONLISP$_0$ definition is based on [Steele Jr., 1984], together with the corrections stated in [Steele Jr., 1990], i.e. on what is commonly referred to as CLtL1.

We will mention extensions and changes proposed by X3J13 in so far as we are able to state whether or not they are *acceptable* for future versions of COMMONLISP$_0$. Proposals for changes or extensions to parts of COMMONLISP which are not intended to belong to COMMONLISP$_0$ will be called *unuseful* throughout this document. In no sense this is meant to value the proposals' quality. However, many constructs of the development environment are unuseful as parts of a subset of the language which is defined for the sake of generating stand alone applications.

Although this document is intended to be a language definition, it will include some remarks stating differences between COMMONLISP$_0$ or CLOS$_0$ and the language which is supported by the current release of our Common Lisp to C Compiler CLiCC. We added these remarks in order to make this document usable as part of the CLiCC documentation.

1.2 A strict subset of COMMONLISP

COMMONLISP$_0$ is intended to be a strict and very large subset of COMMONLISP. By *strict* we mean, that every well defined COMMONLISP$_0$ program is also a well defined and semantically equivalent COMMONLISP program.

The restrictions we defined are different in nature. Some functions like `eval` or `symbol-function` are simply not part of the language COMMONLISP$_0$, others are less defined. Some COMMONLISP functions like `load` are defined to be compile time operations, and hence are restricted to be used at top level of the program with constant arguments only. There are both syntactical and semantical restrictions, and some of the latter aren't even decidable. It might depend on some additional invariants to hold, whether or not a program actually is a well defined COMMONLISP$_0$ program. As an example, the programmer has to ensure that the functional argument *form* in `(apply` *form* `...)` neither evaluates to a symbol denoting a function nor to a list starting with `lambda`. Otherwise, the COMMONLISP$_0$ program would cause an error, whereas the COMMONLISP program eventually runs successfully. Note, however, that this situation does not destroy the property of COMMONLISP$_0$ beeing a strict subset of COMMONLISP.

COMMONLISP$_0$ is intended to be a very large subset of COMMONLISP. It is not our major goal to define a well-structured new lisp-like language. Instead we define COMMONLISP$_0$ as large as the approach of complete compilation reasonably allows, in order to simplify the migration process for real world COMMONLISP applications.

There are some practical advantages which result from COMMONLISP$_0$ beeing a strict subset of COMMONLISP, and which we shall mention here: The programmer may use a conventional Common Lisp system in order to develop, debug, and test COMMONLISP$_0$ applications. Moreover, the strict subset property allows to incrementally migrate COMMONLISP programs into COMMONLISP$_0$, using the development system as a runtime and test environment for the stepwise resulting programs. We refer to [Goerigk and Simon, 1993] for details on migration.

1.3 Notation

As in [Steele Jr., 1990] we will use the `(setf f)` notation in order to refer to the function which *updates* the *place* `f` results in.

In CommonLisp, lists which start with the symbol `lambda` are often implicitely *coerced* to functions. We will call those lists *lambda-lists* throughout this document. Lambda-lists will not be coerced to functions in CommonLisp$_0$. Use `#'(lambda ...)` instead of `'(lambda ...)`. A similar restriction holds for symbols which actually are names of functions. We will often call them *function names*.

2 CommonLisp$_0$

As mentioned earlier, we will not define CommonLisp$_0$ by completely listing the whole language. Instead we will follow the original language definition of CommonLisp [Steele Jr., 1984] chapter by chapter, stating only restrictions and differences of CommonLisp$_0$ with respect to CommonLisp. Therefore, it will be necessary to read a copy of [Steele Jr., 1984] or [Steele Jr., 1990] at the same time in order to obtain the complete definition.

The same strategy will be used in order to define CLOS$_0$. However, the CLOS$_0$ definition is subject of a separate chapter (4) of this document.

Some of the restrictions defined below are of syntactical nature. Some of the CommonLisp language constructs, for example, are restricted to be used as top level forms, only. Please note, that in the following we assume macro calls of the original program to be sufficiently expanded in order to make those constructs visible.

The restrictions defined for CommonLisp$_0$ roughly fall into four classes:

The language constructs which are only useful during the program development process, like `trace`, `break` or `compile-file`, are completely dropped from the language. We recommend the use of a conventional CommonLisp system for the development, debugging and test of CommonLisp$_0$ applications. Hence, the dropped functionality is part of the development environment, but it needs not to be part of the application itself.

We completely disallow *reflection*. Functions which dynamically change the program code, like `eval` and `(setf symbol-function)` are not part of CommonLisp$_0$. Unlike CommonLisp, CommonLisp$_0$ does not implicitely *coerce* symbols or lists starting with `lambda` to functions. Hence, functions like `apply` or `funcall` are restricted to be undefined, if the functional argument actually is a symbol or a list. Multiple definitions of functions and/or macros are not allowed.

In order to generate small application programs it is useful to be able to statically determine the set of potentially called functions within a program. Practically, this would be impossible if there were nontrivial calls to `symbol-function`. A very simple and in some sense necessary method in order to avoid these complications is to avoid `symbol-function` at all. We additionally benefit from this restriction, because for many functions, actually for all those which are not explicitely used as functional objects, it now becomes possible to statically determine all function calls. This will enable us to use classical interprocedural program analysis and optimization techniques which rely on the knowledge of all calling contexts.

In order to enable separate compilation of Lisp modules, we restricted the use of the CommonLisp package operations to really express modularization. The complex functionality of the CommonLisp package system seems to be useful only to avoid name conflicts while reloading or recompiling program parts. And this, again, facilitates the program development process, but needs not necessarily to be part of the application itself.

Note:
The current release of CLiCC does not support the complete set of functions as they are defined to be part of CommonLisp$_0$. *Moreover, some functions, like* `format`*, are not implemented completely. We will document this in future versions. At present, please have a look at the runtime system source code, i.e. the file* `format.lisp` *for the function* `format`*.*

CLiCC will support a foreign function interface to C in one of the future releases which will not be part of CommonLisp$_0$ *as a strict subset of* CommonLisp*, and hence will be documented separately.*

Note, that the titles of the following sections are equal to those used in [Steele Jr., 1984].

2.1 Introduction

No changes.

2.2 Data Types

The CommonLisp data type `function` is restricted to contain "real" functions only, i.e. functions which are the result of evaluating the special form `function`. Function names, i.e. symbols which name a function, and lambda-lists are not functions and will not be implicitely coerced to functions. The range of system functions which deal with functional arguments, like `apply` or `mapcar`, is restricted to the CommonLisp$_0$ data type `function`. They are listed in sections 2.7, 2.14, and 2.15. This restriction has already been proposed by X3J13.

Note:
In the released version, CLiCC does not support bignums, rational and complex numbers, and bit vectors. Moreover, only some of the trigonometric functions are supported.

X3J13 proposed to add the data types `base-character` und `extended-character` instead of the type `extended-character`, which is acceptable. The same holds for the proposed extensions to stream types.

2.3 Scope and Extent

No changes.

2.4 Type Specifiers

Type specifiers are classified to be part of the program text in CommonLisp$_0$. Therefore, we disallow type specifiers to be evaluated at runtime. This would be the interpretation of data as code at runtime, hence *reflection*.

Type specifiers may occur as part of declarations, where they are completely known at compile time. CommonLisp$_0$ does not restrict type declarations.

In CommonLisp, type specifiers are also allowed to be the result of argument forms of functions like `coerce`, `typep`, `subtypep`, `make-sequence`, `concatenate`, `map`, `merge`, `make-array`, `adjust-array`, and `open`. Consequently, we have to restrict their type specifier arguments:

- `coerce`, `typep`, and `subtypep` are defined to be macros. The type specifier arguments are required to be constants.
- `make-sequence`, `concatenate`, `map`, `merge`, `make-array`, `adjust-array`, and `open` are restricted to be defined only on special symbols and lists which denote known types. For further details see the sections 2.14, 2.17, 2.18 und 2.22.
- `type-of` is not part of CommonLisp$_0$. It evaluates type specifiers at runtime and hence has become obsolete in CommonLisp$_0$.

Note, that the first item above implies, that `coerce`, `typep`, and `subtypep` cannot be used directly as functional objects in CommonLisp$_0$. But, of course, functions like `#'(lambda (x) (coerce x '(vector float)))` are allowed.

Since `type-of` is not part of CommonLisp$_0$, forms like `(eq (type-of x) 'integer)` have to be expressed like `(integerp x)`

The restrictions above imply that in CommonLisp$_0$ type specifiers, which are defined using `deftype` forms, can be expanded at compile time. Hence, `deftype` has become syntactic sugar in CommonLisp$_0$.

The functions `upgraded-array-element-type` and `upgraded-complex-part-type` proposed by X3J13 are acceptable with similar restrictions. `coerce`'ing a symbol to a function explicitely, as proposed by X3J13, is just another notation for `symbol-function`, and hence is unuseful.

X3J13 proposed to handle type specifiers uniformly, no matter whether they occur as part of a declaration or as part of a method discriminator, and to add `eql` type specifiers. Both proposals are acceptable. Moreover, the proposed change of the semantics of the `function` type specifier is acceptable. It would allow type inference to actually use the information contained in type declarations, which, in turn, would improve the results of type inference for CommonLisp and CommonLisp$_0$ programs dramatically. We hope this to be part of the ANSI CommonLisp specification and will then adopt it for future versions of CommonLisp$_0$.

2.5 Program Structure

Calls of the CommonLisp macros `defun`, `defvar`, `defparameter`, `defconstant`, `deftype`, `defsetf`, `define-setf-method`, `define-modify-macro`, `defmacro`, and `defstruct` as well as the CLOS$_0$ macro calls to `defclass`, `defgeneric`, and `defmethod` are called to be *defining forms* in CommonLisp$_0$. They are restricted only to occur as *extended top level forms*, which we define as follows:

> **Note:**
> *At present, CLiCC will only accept* `(progn ...)` *forms as extended top level forms. Adding the others is part of our TODO list.*

Definition:

1. Every top level form is an *extended top level form.*
2. If a call of the special forms `progn`, `let`, `let*`, `labels`, `flet`, `macro-let`, `eval-when`, or `progv` is an *extended top level form*, then so are the forms occuring within the body.

X3J13 proposed to add the special forms `symbol-macrolet` and `locally`. This is acceptable. Moreover, they could be added to the list of special forms mentioned in the above definition.

Note, that the above restrictions on the occurance of defining forms allow defining forms to occur within non empty lexical environments. The X3J13 proposal to allow defining forms to occur in arbitrary contexts is unuseful. The reason is, that it has to remain decidable whether or not, and when, a definition actually takes place. Moreover, CommonLisp$_0$ does not allow any redefinition:

Global Macros and global functions share the same namespace, in which multiple definitions are rejected. Note, that this restriction also concerns implicitely defined global functions, like those defined by `defstruct`, and generic functions.

Likewise, constants and global variables, i.e. those names defined by `defconstant` and `defvar`, resp., also share a common namespace. Multiple `defvar`'s are allowed, but constants may not be redefined to be global variables, and vice versa. Multiple definitions of the same constant are rejected.

The fact, that redefinitions are rejected in general, implies, of course, that predefined functions of the `"LISP"` package may not be redefined, as proposed by X3J13. The same holds for predefined constants of the `"LISP"` package.

A restricted version of `eval-when` is part of CommonLisp$_0$. The forms which have to be evaluated at compile time are restricted to be part of the language CL$_{macro}$ which slightly differs from CommonLisp$_0$ and is defined in section 3.

Note:
CLiCC does not support `eval-when` *at present. This is part of our TODO list.*

X3J13 proposed to add `(setf f)` forms as legal function names. This is acceptable. The proposed changes of `eval-when` are acceptable, as well as the proposal to allow arbitrary symbols to denote *keywords* in formal parameter lists.

2.6 Predicates

The operators `typep` and `subtypep` are defined to be macros in CommonLisp$_0$. A macro call `(typep` *object type*`)` is completely expanded at compile time into an appropriate runtime system function call. E.g., `(typep x 'symbol)` will be expanded into `(symbolp x)`. Since type specifiers are restricted to be constant in CommonLisp$_0$ (section 2.4), calls to `subtypep` can be replaced by their results. `(subtypep` *type1 type2* `)` will be expanded at compile time into a call of `values`. In order to give an example, `(subtypep 'cons 'list)` will expand into `(values t t)`. This very rigorous simplification will change in future versions of CommonLisp$_0$ with respect to class arguments of `subtypep`.

Note:
At present, `subtypep` *is not supported by CLiCC.*

The changes to `typep` and `subtypep` proposed by X3J13 are acceptable.

The predicate `functionp` is restricted to be undefined on symbols and lists. The reason is to preserve the strict subset property, i.e. `functionp` may not be defined to yield a different result than the CommonLisp `functionp` eventually would do if applied to a lambda list or function name. Note, that this restriction to `functionp` enlarges the set of non well defined CommonLisp$_0$ programs. This is the price we have to pay for preserving the strict subset property. However, the function `functionp` as proposed by X3J13, is acceptable and could be part of CommonLisp$_0$ without any restrictions.

Note:
Hence, `(functionp 'A)` *should signal an error, which is not the case in our present CLiCC release. Fixing this bug is part of our TODO list.*

2.7 Control Structure

The functions `symbol-function` and `(setf symbol-function)` are not part of CommonLisp$_0$, nor are `fboundp` and `fmakunbound`. The reason for eliminating `(setf symbol-function)` needs no further explanation. We eliminated `symbol-function` as well in order to enable dead code elimination, and to keep the set of eventually called functions small. In CommonLisp$_0$, we know every function which eventually is called, since it has to be mentioned explicitely within the program text, either in a function call or in the form `#'f`.

The function `special-form-p` is not part of CommonLisp$_0$.

The `defsetf` and `define-setf-method` macro bodies are completely evaluated at compile time. The functions `get-setf-method` and `get-setf-method-multiple-value` are useless at runtime of a program and therefore do not belong to CommonLisp$_0$. They will be part of CL$_{macro}$.

Symbols and lambda-lists are not allowed as functional arguments to the functions `funcall`, `apply`, `mapcar`, `maplist`, `mapc`, `mapl`, `mapcan`, and `map`. This is one of the major restrictions in CommonLisp$_0$, and the reason once more is the elimination of reflection: For any form x, the form `(funcall (list 'lambda () x))` is equivalent to `(eval x)`. Instead of lambda-lists or quoted symbols the programmer has to use the forms `#'(lambda (...) ... )` or `#'f`, resp. X3J13 already proposed to eliminate the automatic coercion of lambda-lists to functions, which is acceptable. In CommonLisp$_0$, the coercion of function names to functions is eliminated, too.

X3J13 proposed to add the function `fdefinition`, which essentially is the same as `symbol-function`, and therefore is unuseful.

As proposed already by X3J13, the special form `compiler-let` is not part of CommonLisp$_0$. Adding the special form `symbol-macrolet` would be acceptable.

2.8 Macros

Like in CommonLisp, macro expansion is completely performed at compile time. The language CL$_{macro}$ as defined in section 3, can be used to formulate expressions which have to be evaluated at compile time. It slightly differs from CommonLisp$_0$, but neither CL$_{macro}$ is a subset of CommonLisp$_0$ nor vice versa. Note, that the result of the macro expansion has to fit into CommonLisp$_0$, whereas the forms that compute this result, have to be part of CL$_{macro}$.

The functions `macro-function` and `(setf macro-function)` do not belong to CommonLisp$_0$. Hence, the only way to define a global macro is to use the defining form `defmacro`. A macro has to be defined before its first use. The redefinition of macros is not allowed, and a macro must not redefine a function already defined.

The functions `macroexpand` and `macroexpand-1` do not belong to CommonLisp$_0$, but they do belong to CL$_{macro}$.

Note:
At present, `macroexpand` *and* `macroexpand-1` *are not part of* CL$_{macro}$. *This is part of our TODO list.*

The variable `*macroexpand-hook*` is not part of CommonLisp$_0$.

Within this chapter, the X3J13 proposals mention the additional macros and functions `destructuring-bind`, `define-compiler-macro`, `compiler-macro-function`, `compiler-macroexpand` `compiler-macroexpand-1`, `variable-information`, `function-information`, `declaration-information`, `augment-environment`, `define-declaration`, `parse-macro`, and `enclose`. They are unuseful as parts of CommonLisp$_0$ because it only makes sense to use them during macro expansion. However, as parts of CL$_{macro}$ they are acceptable.

2.9 Declarations

The operator `proclaim` is defined to be a macro in CommonLisp$_0$ which only may occur as an extended top level form (see page 6). Its arguments are evaluated at compile time. The macro `declaim`, as it is proposed by X3J13, is acceptable. But it has to be restricted to only occur as extended top level form.

X3J13 proposed changes with respect to the semantics of type declarations, which are acceptable. They probably will improve the results of type inference for Lisp programs.

2.10 Symbols

No changes.

X3J13 proposed changes with respect to `gensym` and `*gensym-counter*`, which both are acceptable.

2.11 Packages

In CommonLisp$_0$, packages may be used to express *modules*. A CommonLisp$_0$ program then consists of one or more modules. In CommonLisp, on the other hand, packages are used to define separate name spaces for symbols, and the package operations altogether are a powerful tool to maintain these name spaces. At a first glance, these two views of packages have nothing in common. The reason why packages can be used to express modules is in some sense historical: The syntactical representation of a CommonLisp program actually is a list, consisting of a lot of symbols. Since the reader has to be used to read a program text like any other piece of data, symbols denoting program identifiers are modularized into different packages.

Unfortunately, the CommonLisp package operations are too powerful to express a real modularization. The programmer is able to break every module border. The possibility of using "::" package qualifiers or of exporting symbols from different packages, are examples.

The good news is, however, that not every package operation at runtime necessarily has to have an effect to the process of reading the program text itself, i.e. to the compile time environment. Therefore, package operations within a program fall into two classes. Those which do not occur at top level of the program, and hence are not evaluated at compile time, may be used without any restriction. The restrictions which have to hold for the top level or extended top level package operations of a CommonLisp$_0$ program, and which are defined below, are intended to let packages express real modules in the sense of EuLisp or Modula-2.

A module starts with a call of `in-package`. The only legal package top level forms within the subsequent module text are calls of `export`, `import`, `shadow`, `shadowing-import`, and `use-package` with arguments which are evaluated at compile time. A top level package operation must not operate on a package other than the current one. Therefore the optional package argument of the above forms, if supplied, has to denote the current package. Function calls of `make-package`, `unexport`, and `unuse-package` as well as write accesses to the global variable `*package*` must not be used as top level forms. Moreover, none of these forms are allowed within any form of the program which has to be evaluated at compile time. This restriction will remain even if the use of global variables will be allowed in CL_{macro} in future versions of CommonLisp$_0$.

Note:
At present, neither `shadow` *nor* `shadowing-import` *are supported as legal package top level forms. This is part of our TODO list. It will be fixed as soon as the module compilation feature gets released, which, however, is an inofficial part of the current release already* `:-)`

`(in-package ...)` top level forms may occur repeatedly within one module, unless the package argument denotes a different package. This is useful since CommonLisp$_0$ modules may be separated into different files.

A `(load` *filename*`)` top level form loading a file which starts with an `(in-package ...)` form, changing into a new and different package, serves as a module import. The *package*`:`*name* notation may be used to get access to the exported identifiers of the imported module. Of course, `import` or `use-package` may be used to enable this access without any package qualification. However, the *package*`::`*name* notation is not allowed in CommonLisp$_0$.

A module A *depends on* a module B if one or more identifiers of B are referenced within the program text of A. This *module dependency* relation has to form a DAG (directed acyclic graph) in CommonLisp$_0$. This restriction ensures that all of the necessary information about the exported functionality of B is known while analysing A. Hence, the separate compilation of modules is possible, starting with the leaf modules of the module dependency graph of the program.

Note:
The DAG property of the module dependency graph will not be checked by CLiCC in it's current release. This is part of our TODO list.

Within one module, global definitions are restricted to only use names which have the current package as home package. One module must not change the global definitions of another one:

Example:

```
(in-package "a")
  (export 'f)
  (defun f (x) (* x x))
(in-package "b")
  (export 'g)
  (defun g (x) (+ x x))
(in-package "c")
  (use-package "a")
  (f 3)
  (defvar f 4)     ; prohibited !
  (b:g 3)
  (defstruct b:g ) ; prohibited !
```

The changes to the package system proposed by X3J13 are acceptable. The `defpackage` top level form improves the possibility of using packages as modules.

Like `load`, the functions `require` and `provide` are restricted to the extended top level of the program. `require` behaves like `load`, but like in CommonLisp it uses the variable `*modules*` in order to avoid reloading of modules.

2.12 Numbers

No changes.

2.13 Characters

No changes.

2.14 Sequences

Only the symbols `list`, `vector`, `simple-vector`, `string`, `simple-string`, `bit-vector`, or `simple-bit-vector` are allowed to be result type specifications for the functions `make-sequence`, `concatenate`, `map`, and `merge`

The following functions do not accept function names or lambda-lists as functional argument:

`map`, `some`, `every`, `notany`, `notevery`, `reduce`, `remove`, `remove-if`, `remove-if-not`, `delete`, `delete-if`, `delete-if-not`, `remove-duplicates`, `delete-duplicates`, `substitute`, `substitute-if`, `substitute-if-not`, `nsubstitute`, `nsubstitute-if`, `nsubstitute-if-not`, `find`, `find-if`, `find-if-not`, `position`, `position-if`, `position-if-not`, `count`, `count-if`, `count-if-not`, `mismatch`, `search`, `sort`, `stable-sort`, `merge`.

2.15 Lists

The following functions do not accept function names or lambda-lists as functional argument:

`tree-equal`, `push-new`, `subst`, `subst-if`, `subst-if-not`, `nsubst`, `nsubst-if`, `nsubst-if-not`, `sublis`, `nsublis`, `member`, `member-if`, `member-if-not`, `adjoin`, `union`, `nunion`, `intersection`, `nintersection`, `set-difference`, `nset-difference`, `set-exclusive-or`, `nset-exclusive-or`, `subsetp`, `assoc`, `assoc-if`, `assoc-if-not`, `rassoc`, `rassoc-if`, `rassoc-if-not`.

2.16 Hash Tables

The `:test` argument of `make-hash-table` has to be a functional object. Function names or lambda-lists are not allowed.

2.17 Arrays

The functions `make-array` and `adjust-array` are undefined on `:element-type` arguments other than the symbols `t`, `float`, `short-float`, `single-float`, `long-float`, `double-float`, `fixnum`, `bit`, `character`, `standard-char`, `string-char` or lists of the form `(signed-byte` *s* `)` or `(unsigned-byte` *s* `)`.

As proposed by X3J13, the value of `array-dimension-limit` has to be of type `fixnum` in CommonLisp$_0$. The same holds for `array-total-size-limit`.

2.18 Strings

No changes.

2.19 Structures

Redefinitions of structures are not allowed. Name clashes between implicitely generated structure functions and any other global function or macro name will cause an error. Name clashes between the structure name and the name of any other type are rejected as well.

2.20 The Evaluator

CommonLisp$_0$ does not contain the functions `eval`, `evalhook`, `applyhook` and `constantp`, nor does it contain the variables `*evalhook*` and `*applyhook*`.

Since CommonLisp$_0$ is intended to be a language rather than an interactive programming system, the variables `+`, `++`, `+++`, `-`, `*`, `**`, `***`, `/`, `//`, and `///` are not supported. They only make sense in order to control the `read-eval-print` loop of an interactive CommonLisp system.

2.21 Streams

The variables `*debug-io*` and `*trace-io*` are not part of CommonLisp$_0$.

2.22 Input/Output

One of the most dangerous language constructs in CommonLisp is `(eval (read))`. Note, that the input of a form like `(defun f ...)` might even change the running program. The dispatching macros '`#.`' und '`#,`' exactly have this effect, i.e. evaluating the enclosed form while reading it. Therefore they do not belong to CommonLisp$_0$.

2.23 File System Interface

The `:element-type` of the function `open` has to be one of the symbols `string-char`, `character`, `unsigned-byte`, `signed-byte`, or `bit`.

The operator `load` is defined to be a macro, restricted to the extended top level of the program. The arguments are evaluated at compile time. `(load` *file*`)` is substitutet by the content of *file*, or stands for a *module import*, if the situation is as described in section 2.11.

2.24 Errors

CommonLisp$_0$ does not know anything about break loops or interactive error handling. The function `cerror` will ask the user whether or not to continue the program. `break` behaves like `cerror`. `error` aborts the program without any chance to continue.

The variable `*break-on-warnings*` is not part of CommonLisp$_0$.

Since CommonLisp$_0$ is a strict subset of CommonLisp, any existing interactive Common-Lisp system will serve as a more or less comfortable debugging and test environment for CommonLisp$_0$ programs.

2.25 Miscellaneous Features

The functions `compile`, `compile-file`, `disassemble`, and `documentation` do not belong to CommonLisp$_0$.

`trace`, `untrace`, `step`, `time`, `describe`, `inspect`, `room`, `ed`, `dribble`, `apropos`, and `apropos-list` are definitely part of the development environment of CommonLisp systems, and therefore, are not part of the language CommonLisp$_0$.

3 CL$_{macro}$

Like CommonLisp$_0$, CL$_{macro}$ is a subset of CommonLisp. But neither CL$_{macro}$ is a subset of CommonLisp$_0$ nor vice versa. Expressions of a CommonLisp$_0$ program, which have to be evaluated at compile time, like macro bodies, have to belong to CL$_{macro}$. Expressions, which eventually are evaluated at compile time and at runtime, like bodies of functions which are used to compute macro expansion results, have to satisfy the restrictions defined for both CommonLisp$_0$ and CL$_{macro}$.

The definition of CL$_{macro}$ is not yet finished. One of the open questions at present is whether or not we should allow global variables to be interpreted at compile time. We are aware of applications which use global variables for macro expanders to communicate with each others. But there are problems with the distinction between compile time and runtime values of those variables. They may vary if the compiler does not evaluate the whole program at compile time already. The problems are wellknown, and many programmers have had the same problems while writing portable CommonLisp code.

The functions `get-setf-method`, `macroexpand`, and `macroexpand-1` are part of CL$_{macro}$, but they do not belong to CommonLisp$_0$, since accessing the compile time environment makes no sense at runtime.

Note:
At present, CL$_{macro}$ *is a subset of* CommonLisp$_0$ *because the additional features are not yet implemented.*

CL$_{macro}$ does not include any package operations. Note, that e.g. macro expansion might result in package operations, but must not perform any package manipulation. Every package operation evaluated at compile time has to occur explicitely as a legal package top level operation, after sufficiently expanding top level macro calls, though.

CommonLisp does not specify anything about how a reasonable compile time environment for evaluating forms at compile time looks like. For good reasons, it is not defined whether or not *compiling* a definition also means *evaluating* that definition within the compile time environment, although many CommonLisp systems will do so. Consequently, nothing is defined for the compile time environment for CommonLisp$_0$.

4 CLOS$_0$

Like CommonLisp$_0$, which essentially is defined in order to exclude *reflection* from CommonLisp, CLOS$_0$ will be defined as a subset of CLOS, which essentially excludes unrestricted uses of reflection. The reason again is, that we want to ensure the quality of *global* program analysis and optimization for CommonLisp$_0$ application programs which use object oriented programming.

But, unlike reflection in CommonLisp, in CLOS the notion of reflection has various aspects. First of all, the distinction between program and data is not as simple and wellknown for the object system as for the non object oriented kernel of CommonLisp. Instances are data, of course. But should classes, i.e. types, be program or data? Should classes be instances of metaclasses? At present, there are many different answers to questions like this. The intention of CLOS$_0$ is to call those operations, which work on classes or generic functions as data objects, *reflective*. Hence, they are excluded from the language.

On the other hand, metaobject programming, at least in a restricted manner, seems to be very useful even for application programming. At least some parts of the metaobject protocol should be supported by CLOS$_0$, whereas other parts, like class redefinition, definitely belong to the development environment. Although some reflection should be part of CLOS$_0$, at present it is not. We have to be very careful in improving the present definition in this aspect, because we have to ensure both, the strict subset property, and the quality of static program analysis.

The strategy we will use to improve CLOS$_0$ in order to include metaobject programming will be to keep those operations apart from runtime, like runtime macro expansion has been removed from the modern Lisp language versions.

The definition on CLOS$_0$ included here has to be taken as a first attempt in order to define a suitable subset of CLOS. At present, the restrictions are very rigorous in order to achieve our goals: There is no meta object protocol. Classes, generic functions and methods are program, not data. There is no possibility to define or redefine classes, methods, or generic functions at runtime, nor to add methods to generic functions. Changing an instance's class is impossible. There are objects representing classes, but they have no class. It is an error to access a class's class. Moreover, CLOS$_0$ ensures class objects only to represent classes which are statically defined within the program text.

Again, not every useful CommonLisp + CLOS program belongs and will belong to the subset defined within this document. But we hope that it will be rich enough to express the most requirements of Lisp application programs.

4.1 Programmer Interface Concepts

As before, we will follow the section headings of the original specification, i.e. the chapter 28 of [Steele Jr., 1990], in order to define CLOS$_0$ by stating changes and differences with respect to the original. Section 4.2 then will summarize both the functionality that is supported by CLOS$_0$ together with the restrictions defined, and those functions, macros, or generic functions which are not.

Restrictions are defined both for the *metaobject protocol* and the *functional interface*, which CLOS provides in order to define classes, generic functions, and methods at runtime. Therefore, in CLOS$_0$ the only legal method to create classes, generic functions or methods is to use `defclass`, `defgeneric`, or `defmethod`, which are *defining forms* and hence are restricted to occur as extended toplevel forms only. (cf. section 2.5)

The generic functions which are classified to be part of the metaobject protocol, are:

- `shared-initialize`
- `update-instance-for-different-class`
- `update-instance-for-redefined-class`
- `compute-applicable-methods`
- `ensure-generic-function`

- `compute-effective-method`
- `allocate-instance`
- `make-instances-obsolete`

They do not belong to $CLOS_0$. As a consequence, many functions and generic functions which remain in $CLOS_0$, like `initialize-instance`, actually do behave as this is defined for the standard situation, but their behavior is not customizable using method definitions for generic functions of the metaobject protocol.

We classify `add-method`, `remove-method`, `generic-function`, and calls to *make-instance* with metaclasses as class arguments to belong to the functional interface. $CLOS_0$ does not support them. Moreover, $CLOS_0$ does not support metaclasses at all. Thus, there is no way to create instances of metaclasses, like anonymous classes or generic functions, nor to dynamically change or enlarge the set of methods which belong to a generic function.

The lists above are intended to give a first impression of the nature of the restrictions defined within this document. Hence, they are not complete, i.e. there are more important restrictions, and more functions, like `change-class`, are excluded, although they are neither part of the metaobject protocol nor do belong to the functional interface. We refer to the subsequent sections for details.

4.1.1 Classes

The classes `standard-class`, `standard-generic-function`, and `standard-method` are metaclasses in CLOS. Therefore, they do not belong to $CLOS_0$.

Unlike in CLOS, `find-class` is restricted to be defined on constant symbols which name user defined classes, i.e. a classes which are defined using `defclass`.

The function `make-instance` is not generic. It is restricted to be defined only on class objects, which, in turn, only can be accessed using `find-class`. Hence, $CLOS_0$ ensures class objects only to represent classes which are defined using the extended toplevel form `defclass` and which, therefore, are statically known from the program text. In order to avoid unnecassary syntactical transformations, `(make-instance '`<*name*>` ...)` internally is transformed to `(make-instance (find-class '`<*name*>`) ...)`, and hence, is legal in $CLOS_0$.

The `defclass` macro does not support the class option `:metaclass`.

Moreover, the `:allocation` slot option must not override the `:allocation` option of an inherited slot.

The standard methods defined for `slot-unbound` and `slot-missing` will signal and uncontinuable error and hence will abort the program execution. Note, that, according to the current definition of $CLOS_0$, there is no way to define methods for `slot-unbound` and `slot-missing` because they would have to be specialized over metaclasses. This is true for `no-applicable-method` and `no-next-method` as well (cf. section 4.1.6). Hence, an implementation can choose to call simple error functions instead of these generic functions.

4.1.2 Inheritance

The `:allocation` slot option is restricted not to override specified `:allocation` options of inherited slots. Thus, unlike in CLOS, shared slots cannot be shadowed in CLOS$_0$.

4.1.3 Integrating Types and Classes

The functions `typep`, `type-of`, and `subtypep` are restricted according to the sections 2.6 and 2.4 of the CommonLisp$_0$ part of this document. As mentioned there already, these very rigorous restrictions will be weakened with respect to classes in future versions of CommonLisp$_0$ + CLOS$_0$.

The metaclasses `built-in-class` and `structure-class` do not belong to CLOS$_0$.

4.1.4 Determining the Class Precedence List

No changes.

4.1.5 Generic Functions and Methods

As mentioned earlier, the *functional interface* in CLOS is intended to enable runtime creation and changes to classes, generic functions and methods. It does not belong to CLOS$_0$. Hence, only the extended top level forms `defgeneric` and `defmethod` can be used in order to create generic functions or to define methods. Moreover, CLOS$_0$ does not support anonymous global generic functions.

Consequently, `add-method`, `remove-method`, `with-added-methods` as well as the macro `generic-function` do not belong to CLOS$_0$. Neither `with-added-methods` not the special forms `generic-flet` and `generic-labels` do belong to the ANSI CommonLisp proposal. The present definition of CLOS$_0$ does not support local generic functions, either. According to the results of the ANSI CommonLisp discussion, `generic-flet` and `generic-labels` might be added to future versions of CLOS$_0$.

The `defgeneric` and `defmethod` defining forms do not support the options `:generic-function-class` and `:method-class`.

With respect to CommonLisp$_0$ modules, there is one more restriction which has to hold: If methods for one generic function *f* are to be defined within two different modules, the appropriate generic function definition for *f*, either explicitly or implicitely using `defmethod`, has to occur within one common imported module.

4.1.6 Method Selection and Combination

Only the standard method combination and the *simple built in* method combination types are available in CLOS$_0$. The macro `define-method-combination` is not part of CLOS$_0$. Hence, the `:method-combination` option to `defgeneric`, if supplied, is restricted to the constant keys `:+`, `:and`, `:append`, `:list`, `:max`, `:min`, `:nconc`, `:or`, `:or`, and, of course, `:standard`, which is the default. Although method combinations will not become *meta* *objects*, the macro `define-method-combination` might become part of future versions of CLOS$_0$.

Calls to `call-next-method` must not specify arguments. In CLOS$_0$ the set of next methods will be determined statically. If `call-next-method` were called with incompatible arguments, the chain of next methods could be broken and hence could change at runtime. This is a wellknown problem in CLOS which the definition of CLOS$_0$ intends to avoid.

The generic functions `no-applicable-method` and `no-next-method` will signal an uncontinuable error and hence will abort the program execution. (Cf. section 4.1.1.)

The functions `invalid-method-error` and `method-combination-error` as well as the macro `call-method` do not belong to CLOS$_0$.

4.1.7 Meta-objects

CLOS$_0$ does not support any of the *meta objects* built into CLOS, nor does it support *metaclasses*. Therefore, creating meta objects is impossible in CLOS$_0$. It is an error to apply `class-of` to classes, generic functions, or methods. The function `class-of` is restricted only to be defined on instances of user defined classes. Moreover, method combinations are not objects, and, therefore, do not belong to a class either.

4.1.8 Object Creation and Initialization

In CLOS$_0$, the function `make-instance` is not generic. It is restricted only to be defined on class objects, which, in turn, only can be created using `find-class` applied to a constant symbol which has to denote a defined named class. `(make-instance '<name> ...)` is a syntactical abbreviation for `(make-instance (find-class '<name>) ...)`.

The generic functions `shared-initialize` and `allocate-instance` do not belong to CLOS$_0$. Thus, CLOS$_0$ defines the standard behavior as specified for CLOS to be the semantics of `make-instance`. The one and only exception is that `initialize-instance` might be customized using specialized method definitions. With respect to CommonLisp$_0$ modules, one more restriction has to hold: If a `(defclass` A `...)` defining form occurs within one module, all of the method definitions specializing `initialize-instance` over A have to be defined within the same module.

Although `reinitialize-instance` does belong to CLOS$_0$, it will not be called implicitely. The reason is that neither class redefinitions nor changing the class on an instance, i.e. `change-class`, are allowed.

The generic functions `update-instance-for-redefined-class` and `update-instance-for-different-class` do not belong to CLOS$_0$.

4.1.9 Redefining Classes

We classify class redefinitions to belong to the program development interface and, thus, they are rejected in CLOS$_0$. The complete metaobject protocol defined in this section of [Steele Jr., 1990] does not belong to CLOS$_0$. Class redefinition will not be supported in future versions of CLOS$_0$, whereas some parts of the metaobject protocol, like `shared-initialize`, might become part of future versions.

4.1.10 Changing the Class of an Instance

The function `change-class` and the complete metaobject protocol defined here are not, and will not be supported by CLOS$_0$. The reason is that we want to ensure the correctness of type inference results, which, of course, would be impossible with respect to objects eventually changing their classes.

One possible way in order to weaken this restriction is to allow a *monotonic* version of `change-class`, i.e. changing the class of an instance only to more specific classes. But even this restricted version would disable many optimizations which rely on the concrete knowledge of the direct class of an instance. Hence, we decided to drop `change-class` at all.

4.1.11 Reinitializing an Instance

Since `shared-initialize` does not belong to CLOS$_0$, the generic function `reinitialize-instance` does not call `shared-initialize`.

4.2 Functions in the Programmer Interface

The following two lists are intended to summarize those operators, which are, and those, which are not part of CLOS$_0$.

- `call-next-method`
  Restricted. Cf. section 4.1.6 for details.
- `class-of`
  Restricted. Cf. section 4.1.7 for details.
- `class-name`
- `defclass`
  Restricted. Cf. section 4.1.1 for details.
- `defgeneric`
  Restricted. Cf. section 4.1.5 and section 4.1.6 for details.
- `defmethod`
  Restricted. Cf. section 4.1.5 for details.
- `find-class`
  Restricted. Cf. section 4.1.1 and section 4.1.8 for details.
- `initialize-instance`
  Restricted. Cf. section 4.1.8 for details.

- `make-instance`
  Restricted. Cf. section 4.1.8 for details.
- `next-method-p`
- `no-applicable-method`
  Restricted. Cf. section 4.1.6 for details.
- `no-next-method`
  Restricted. Cf. section 4.1.6 for details.
- `print-object`
- `reinitialize-instance`
  Restricted. Cf. section 4.1.11 for details.
- `slot-boundp`
- `slot-makunbound`
- `slot-missing` Restricted. Cf. section 4.1.1 for details.
- `slot-unbound` Restricted. Cf. section 4.1.1 for details.
- `slot-value`
- `with-accessors`
- `with-slots`

The following list summarizes those operators which do not belong to CLOS_0:

– `add-method`

– `call-method`

– `change-class`

– `compute-applicable-methods`

– `define-method-combination`

– `documentation`

– `ensure-generic-function`

– `find-method`

– `function-keywords`

– `generic-flet`

– `generic-funtion`

– `generic-labels`

- `invalid-method-error`
- `make-instances-obsolete`
- `method-combination-error`
- `remove-method`
- `shared-initialize`
- `slot-exists-p`
- `update-instance-for-different-class`
- `update-instance-for-redefined-class`
- `with-added-methods`

5 Acknowledgements

We would like to thank Harry Bretthauer and Jürgen Kopp for an intensive discussion and many suggestions helping us to improve the definition of CLOS$_0$.

References

[Burkart, 1991] O. Burkart. Das Frontend eines Compilers von Common Lisp nach C. Master's thesis, Institut für Informatik und Prakt. Math. der Christian-Albrechts-Universität, Kiel, 1991.

[Goerigk and Simon, 1993] W. Goerigk and F. Simon. Migration und Kompilation in Lisp: Ein Weg von Prototypen zu Anwendungen. APPLY-Arbeitspapier APPLY/CAU/IV/2, Christian-Albrechts-Universität, Kiel, May 1993.

[Knutzen, 1991] H. Knutzen. Codegenerierung und Laufzeitsystem eines Compilers von Common Lisp nach C. Master's thesis, Institut für Informatik und Prakt. Math. der Christian-Albrechts-Universität, Kiel, 1991.

[Kopp *et al.*, 1991] J. Kopp, J. Beste, H. Bretthauer, T. Christaller, H. Friedrich, W. Goerigk, W. Heicking, D. Hovekamp, H. Knutzen, U. Kriegel, I. Mohr, R. Rosenmüller, and F. Simon. APPLY. Ein bedarfsgerechtes und effizientes Lisp für Anwendung und Forschung. Antrag zu einem Verbundvorhaben, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Sankt Augustin, January 1991.

[Steele Jr., 1984] G. L. Steele Jr. *Common Lisp - The Language.* Digital Press, Bedford, Massachusetts, 1984.

[Steele Jr., 1990] G. L. Steele Jr. *Common Lisp - The Language, Second Edition.* Digital Press, Bedford, Massachusetts, 1990.