CLICC: A New Approach to the Compilation of Common Lisp Programs to C

O. Burkart W. Goerigk H. Knutzen
Institut für Informatik und Praktische Mathematik
der Christian-Albrechts-Universität zu Kiel
Preußerstr. 1-9, D-2300 Kiel
wg@informatik.uni-kiel.dbp.de, wg@causun.uucp

Abstract

We describe an implementation technique for Common Lisp application programs on a large variety of small computers. A *strict* subset of the programming language Common Lisp, called Common Lisp$_1$, is defined, and we describe a compiler from Common Lisp$_1$ to C, called CLICC, which we successfully used to build up stand alone Common Lisp applications of reasonable size and efficiency, including expert system applications and CLICC itself. Common Lisp$_1$ programs are portable on the basis of machine independent C source code. We propose Common Lisp$_1$ as an efficiently implementable strict subset of Common Lisp. The notion of a Common Lisp$_1$ *application program* is proposed in order to enlarge the compilation units and to make global program optimizations possible.

Contents

1 Introduction

During the past decades the programming language LISP ([McC60], [MAE+62]) has been developed into a large programming tool supporting different programming paradigms. Among

many other LISP dialects, Common Lisp ([Ste84], [Ste90]) has become a de facto standard for LISP in industry and science. It is commonly used in artificial intelligence, computer algebra, compiler construction, and theorem proving. Up to now Common Lisp applications need large and expensive development or runtime systems on large machines to run. The major goal of the work described here is to show how the compilation of a strict subset of Common Lisp to C can be used to run application programs on small machines without a Common Lisp system. We propose a distinction between the language constructs needed for application programs and those needed for incremental program development within an interpreter based Common Lisp system.

CLICC, the compiler from Common Lisp to C, is able to compile a strict subset of Common Lisp into C source code. We call this subset Common Lisp_1 (CL_1) and assume it to be the target language of a program development process within a Common Lisp system. Our approach differs from the usual incremental compilation used in Common Lisp systems. We call our technique *complete compilation of programs* because it is based on a complete set of informations about the entire program.

In addition to restricting programs not to use the whole functionality of the program development system, we decided not to support Common Lisp's *reflection* in order to keep the compiled code small and to allow global program optimization. Thus, explicit calls of `EVAL` as well as language constructs which implicitely cause evaluation of data as code at runtime, like `MACROEXPAND` or `SYMBOL-FUNCTION`, are not allowed in Common Lisp_1 programs. User defined macros are syntactically restricted to be expandable at compile time. Moreover, double declarations and incomplete code, i.e. calls of nondefined functions, are rejected.

The target language for CLICC is C [KR83] which we use as a portable machine oriented language. Common Lisp_1 functions are compiled into C functions which play the role of low level subroutines. Parameter passing is implemented on a *LISP stack* to enable garbage collection. C parameters are used to hold linkage information which is needed to correctly implement both, nested function definitions and higher order functions. Both of them are not directly supported by the C procedure concept.

CLICC ([Knu91], [Bur91]) has been developed [1] at the University of Kiel during the past two years. It was used as a backend for a compiler from BABYLON[2] to C using Common Lisp as an intermediate language. We compiled knowledge bases which have been developed within the BABYLON workbench into portable stand alone expert system applications.

In section 2 the language Common Lisp_1 is defined as a strict subset of Common Lisp. Many of the program constructs of Common Lisp_1 including system macro calls, are defined in Common Lisp_1 itself. Thus, the first compilation step is a source to source code transformation of Common Lisp_1 into an intermediate or kernel language which we call Common Lisp_0 (CL_0). Global program analysis is performed on an abstract syntax representation of Common Lisp_0 programs to support efficient code generation. The expansion of system macros and user defined macro calls, the global program analysis, and the code generation are described in section 3. The complete set of Common Lisp_1 system macros and their expansion rules are added as appendix B.

Some Common Lisp_1 standard functions are handcoded in C, others are implemented in LISP. Thus, the Common Lisp_1 runtime system consists of several C source code parts, some of which are produced by CLICC itself. In order to generate executable application programs, all of those parts together with the C target code of the original program have to be compiled

[1] The work was done in a project called "*knowledgebase compilation*" ([AG87], [AGS88]). The project was sponsored by the Dr.-Ing. Rudolf Hell GmbH, Kiel.

[2] BABYLON is the hybrid knowledge representation language of the expert system tool BABYLON, which was developed as an AI workbench at the GMD, St.Augustin.

into object code by the C compiler and linked together with the linker of the target system. This process is described in section 4.

CLICC is implemented in Common Lisp$_1$. Thus, CLICC is able to compile itself and, therefore, like any Common Lisp$_1$ application program, it is portable on the basis of C source code. Portablility and the wellknown compiler bootstrap test are the subject of section 5.

2 Definition of Common Lisp$_1$

We call our approach *complete compilation of programs*. It differs from the usual incremental approach because the compilation is based on a complete set of informations about the entire program at compile time. Our major goal is to compile Common Lisp *application programs* into efficient C programs which are executable without a Common Lisp system and thus are portable on the basis of C source code. In order to achieve efficiency and portability the compiled Common Lisp language has to be restricted. The subset Common Lisp$_1$ (CL$_1$) of Common Lisp which fulfils the intended purpose is defined in this section. CL$_1$ is intended to be our proposal for *the programming language Common Lisp*. CL$_1$ differs from Common Lisp as follows:

- CL$_1$ does not allow interpreter calls at runtime of programs.

For the sake of efficiency we do not support the interpretation of data as programs at runtime. Therefore all functional objects which might be called within a program, are known at compile time. Thus, the compiler is enabled to perform global program optimizations.

- Symbols do not have a *function cell*.
- Double declarations and incomplete code are rejected.

Usually LISP systems use the function cell of a symbol to store the global function definition named by the symbol. Without the function cell of a symbol and without redefinitions of functions at runtime the function bindings can be completely determined at compile time. This allows direct function calls without using the function cell (*block compilation* [Ste90]).

- User defined macros are completely expanded at compile time.

User defined macro bodies are syntactically restricted to be *simple* `BACKQUOTE`-*forms* as they are defined below. Runtime macro expansions, which would cause in interpreter calls at runtime, are not allowed. There is no need of linking macro expansion functions into the compiled code.

- Forms that have to be evaluated at compile time are restricted to be *simple* `BACKQUOTE`-*forms*.

The compilation of certain forms involve the interpretation of data as programs at compile time. Those forms are restricted to be *simple* `BACKQUOTE`-*forms*, i.e. forms which CLICC as a CL$_1$ program is able to evaluate at compile time.

2.1 Syntax and Context Conditions

The subject of this section is to give an overview of the syntax of Common Lisp$_1$. We also describe the context conditions which have to hold for CL$_1$ programs in addition to those adopted from the Common Lisp language definition. The complete syntax definition is given in appendix A. The set of implemented data types and standard functions is described in section 2.2. Appendix B defines the implemented system macros and their expansion rules.

We propose the notion of a Common Lisp$_1$ *application program* which is defined by

cl$_1$-program ::= { *top-level-form* }*

where *top-level-forms* are defined by

top-level-form ::=
(`IN-PACKAGE` *package-name* `&key :nicknames :use`) |
(`LOAD` *file-name*) |
...
(`DEFVAR` *name* [*initial-value* [*documentation*]]) |
...
(`DEFMACRO` *name bq-lambda-list* [*doc-string*] *bq-form*) |
(`DEFUN` *name lambda-list* [[{ *declaration* }* | *doc-string*]] { *form* }*) |
...
form.

A Common Lisp$_1$ program is defined as a sequence of top level forms which can be devided into function *definitions*, *declarations*, and *expressions* or *statements*. Code is generated by the compiler for function definitions via (`DEFUN` ...) and top level *form*'s. Pure declarations like the *special declaration* (`DEFVAR` *name*) just change the compile time environment, e.g. (`DEFVAR` *name*) globally declares *name* to denote a dynamically scoped variable. In general, most of the declarations are top level *statements*, too, e.g. (`DEFVAR` *name initial-value*) means the pure declaration as well as the top level assignment of the result of *initial-value* to *name*.

Declarations are collected as annotations to the abstract representation of the program or change the compiletime environment, whereas top level expressions and statements make up the main part of the program.

Redefinitions and *redeclarations* of functions, macros, types, `SETF`-methods, and structures are rejected. Especially system functions must not be redefined.

Global *special declarations* for a variable must occur before the first binding of the variable is processed by the compiler.

User defined macros are inserted into the compiler environment. Entries of user defined macros are used to expand macro calls of the form (*name* ...) into the resulting *bq-form* at compile time. Therefore (`DEFMACRO` ...) can be seen as a pure declaration in CL$_1$ and we do not have to generate code for macro expansion functions. Init forms in macro parameter lists as well as the macro body are restricted to be *bq-forms*:

bq-form ::= *atom* | (`QUOTE` *s-expr*) | *bq-call*
bq-call ::= (`CONS` *bq-form bq-form*) |
(`LIST` { *bq-form* }*) |
(`LIST*` { *bq-form* }*) |
(`APPEND` { *bq-form* }*) |

Macro definitions have to preceed their first use, and variables which occur in a simple BACKQUOTE-form have to be bound by the enclosing lambda expression.

Common Lisp system functions like (LOAD ...) or (IN-PACKAGE ...) are defined to be top level forms in CL_1.

IN-PACKAGE causes different actions at compile time and runtime. At compile time it changes the compiler environment, such that the symbols of the subsequent program text are included into the package *package-name*. At runtime it changes the current package to *package-name*. IN-PACKAGE forms are restricted to have only constant arguments. Package qualifiers to symbols are allowed in CL_1, but the appropriate (IN-PACKAGE ...) form has to preceed the first occurance of a qualified symbol. Sideeffects to the symbol packages of the compiler and runtime system, i.e. CLICC and RUNTIME, are not allowed. Thus, user programs must not use SHADOW, SHADOWING-IMPORT, (UN)EXPORT, (UN)USE-PACKAGE or IMPORT forms which influence one of the packages CLICC or RUNTIME.

(LOAD *file-name*) is replaced by the contents of *file-name* at compile time. Thus, LOAD and REQUIRE are restricted to be used only as top level forms which are expanded syntactically. PROVIDE just changes the compile time environment to record the module provided.

2.2 Data Types and Standard Functions

We have implemented many of the Common Lisp data types and standard functions. Some of the functions are not yet implemented, others will not be implemented at all since our approach of *complete compilation* classifies them to be part of the programming environment or to cause evaluation of data at runtime. There are some problems to implement bignums in C portably and efficiently. Moreover, most pathname and time functions are not portably implementable. But most of the Common Lisp functions we do not support at present are easily implementable. Up to now only their large number prevented us from doing so.

Data Types

We give an overview of the implemented data types and the corresponding standard functions in the same order as the data types are described in the chapter "DATA TYPES" in [Ste90].

Numbers

Integers with limited precision (fixnums) are implemented using the C data type `long`. Overflows are not detected, e.g (+ 1 MOST-POSITIVE-FIXNUM) results in MOST-NEGATIVE-FIXNUM. It is not possible to implement integers with arbitrary precision (bignums) in a portable and efficient manner in C. All of the functions which deal with integers are implemented. Ratios and complex numbers are not yet implemented, but there are no difficulties to do so. The four possible floating-point types of Common Lisp are implemented as one type using the C data type `double`. It conforms with the Common Lisp standard to make no differences between the floating-point types. One could use the C data type `float` to implement a floating-point type of different precision. We have implemented only a few of the functions which deal with floating-point numbers. Input and output of floating-point numbers is incomplete but usable at present. (E.g. *READ-DEFAULT-FLOAT-FORMAT* is not supported).

Characters

Standard-characters and semi-standard characters are supported. Non-standard characters and character attributes are not implemented. Character attributes are optional in [Ste84]

and removed in [Ste90]. All character functions which do not access character attributes are implemented.

Symbols
Symbols are completely implemented but they do not have a function cell. Therefore `SYMBOL-FUNCTION, FBOUNDP, FMAKUNBOUND, SPECIAL-FORM-P` are not implemented.

Lists and Conses
Lists and Conses are completely implemented.

Arrays
Arrays are implemented and we support specialized arrays for characters, fixnums and floats. Specialized arrays of bits are not yet implemented and therefore the data type bit-vector is not supported. Arrays with fill pointers, displaced arrays and adjustable arrays as well as the corresponding functions are implemented.

Hash-Tables
Hash-tables are not yet implemented.

Readtables
Readtables are completely implemented. It is possible to define macro characters. We have implemented the function `SET-SYNTAX-FROM-CHAR` to be applicable to macro characters too, which causes some problems in other LISP implementations. We have not yet implemented all of the predefined read macros.

Packages
Packages are implemented as described in [Ste84]. We did not consider changes and extensions of [Ste90] yet. The package functions are implemented too, but only `IN-PACKAGE` is recognized by CLICC as a toplevel *declaration* during the compilation process.

Pathnames
Pathnames are not yet implemented. It is not possible to implement them in a portable way, because the corresponding functions are highly operating system dependent. At present we use strings instead of pathnames in functions like `OPEN`.

Streams
Only character streams are implemented at the moment. The stream `*TERMINAL-IO*` and the streams which are generated by the function `OPEN` are links to file descriptors of the underlying operating system. We use only functions of the C standard library to implement streams and stream functions, hence we can generate portable code which is independent of the operating system.

Random-States
Random-states are not yet implemented.

Structures
Structures are implemented to a great extent. The type specifications of slots are not checked. The structure definition options `:TYPE` and `:INITIAL-OFFSET` and BOA constructors (By Order of Arguments) are not supported yet.

Functions
`FUNCTION` is the only special form which yields functions. Symbols and lambda lists are not of type function and in CL_1 there is no possibilty to convert them into a function. All data objects of type function are compiled functions. All the standard functions which are defined on the type function (e.g. `APPLY`) will abort with an error message, if they are called with a

symbol or lambda list instead of a functional object as argument.

Other Standard Functions

The following gives a description of the functions which are not or only partially implemented and which are not related to data types. The functions are ordered corresponding to the chapters of [Ste90].

Type Specifiers

The function `COERCE` is implemented only partially. The function `TYPE-OF` is not implemented.

Predicates

The function `TYPEP` is expanded at compile time if possible. The runtime version of `TYPEP` is restricted to be applicable to simple type specifiers like `FIXNUM` or `CHARACTER` etc.. The function `SUBTYPEP` is implemented only for a few types. Most of the other predicates are implemented if the corresponding data types are implemented.

Control Structure

The functions `GET-SETF-METHOD` and `GET-SETF-METHOD-MULTIPLE-VALUE` are not and will not be implemented because all `SETF` forms have to be expanded at compile time.

Macros

The functions `MACRO-FUNCTION, MACROEXPAND` and `MACROEXPAND-1` are not implemented because macros are expanded at compile time. These functions will not be implemented in the future.

Sequences

Most of the sequence functions are implemented. We left out some complex functions like `SORT` which may be added without problems.

The Evaluator

The function `EVAL` is not implemented.

Input/Output

All of the input/output functions are implemented, but most of them do not have their full functionality. E.g. the function `FORMAT` does not accept all of the possible control directives.

File System Interface

The functions `OPEN, PROBE-FILE, FILE-POSITION` and `FILE-LENGTH` are implemented. The function `LOAD` is expanded at compile time and may not be used at runtime because we do not support incremental compilation and dynamic loading. All other functions which are described in that chapter are not implemented.

Errors

Only the functions `ERROR` and `WARN` are implemented. The functions `CERROR` and `BREAK` cannot be used, because we do not support break loops.

Miscellaneous Features

Only the function `IDENTITY` is implemented. The functions dealing with the compiler or the debugging tools cannot be used. The functions for environment inquiries could be implemented in the future, but there are problems to implement time functions portably.

3 Compilation of Common Lisp$_1$ Programs to C

CLICC is a multi pass compiler. The frontend uses different passes to compile CL_1 programs into an abstract representation of an intermediate or kernel language which we call Common Lisp$_0$ (CL_0). In a first pass user defined macro calls and system macro calls are expanded using a source to source code transformation. An example is given in section 3.1.

Declarations are collected into compile time data structures which are part of the abstract representation of the program. This second pass is not decribed here. We refer to [Bur91] for more details. In section 3.2 we describe global program analysis. In a final pass C code is generated. The code generation is subject of section 3.3.

3.1 Source Code Transformation

We will demonstrate the expansion of system macro calls according to the expansion rules given in appendix B by the following example. This small part of a CL_1 program iterates over an input list `list`, searching for occurances of the object `search`, and prints their indices, if found.

```
(DO ( (index       0 (1+ index))
      (tmp-list list (cdr tmp-list))
    )
   ((null tmp-list))
   (WHEN (eq (car tmp-list) search)
      (print index)
      (RETURN))
)
```

The expansion of `DO` results in:

```
(PROG ( (index       0)
        (tmp-list list)
      )
   g001   ;; g001 is a new generated symbol to prevent name clashes
   (WHEN (null tmp-list) (RETURN (PROGN)))
   (WHEN (eq (car tmp-list) search)
      (print index)
      (RETURN))
   (PSETQ index (1+ index) tmp-list (cdr tmp-list))
)
```

The expansion of `PROG` yields:

```
(BLOCK nil
   (LET ( (index       0)
          (tmp-list list)
        )
      (TAGBODY
         g001
         (WHEN (null tmp-list) (RETURN (PROGN)))
         (WHEN (eq (car tmp-list) search)
```

```
            (print index)
            (RETURN))
         (PSETQ index (1+ index) tmp-list (cdr tmp-list))
      )
   )
)
```

A `LET` special form is transformed into the corresponding `LAMBDA` application. The `WHEN` macro calls are expanded into equivalent `IF` forms.

```
(BLOCK nil
   ((LAMBDA (index tmp-list)
         (TAGBODY
            g001
            (IF (null tmp-list)
               (PROGN (RETURN (PROGN)))
               nil)
            (IF (eq (car tmp-list) search)
               (PROGN (print index)
                      (RETURN))
               nil)
            (PSETQ index (1+ index) tmp-list (cdr tmp-list))
         ))
      0 list)
)
```

After expansion of `RETURN` and `PSETQ` and some optimizations this yields the following final result form:

```
(BLOCK nil
   ((LAMBDA (index tmp-list)
         (TAGBODY
            g001
            (IF (null tmp-list)
               (RETURN-FROM nil nil)
               nil)
            (IF (eq (car tmp-list) search)
               (PROGN (print index)
                      (RETURN-FROM nil nil))
               nil)
            (SETQ index
                  ((LAMBDA (g002)   ;; g002 is a new generated symbol
                        (SETQ tmp-list (cdr tmp-list))
                        g002)
                     (1+ index)))
         ))
      0 list)
)
```

3.2 Global Program Analysis

The *complete compilation of programs* enables the compiler to perform *global program analysis*. The goal of such an analysis is to get informations about the behaviour of the program that allow the generation of more efficient code. Up to now there are two global analysis methods implemented in our compiler.

The first concerns the check of passing a correct number of arguments to a function. In general LISP systems have to perform this check at runtime because the arity of a function may change in the development process. In our case the arities of all used functions are known. This fact enables us to perform a partial check of correct parameter passing at compile time. Parameter passing can only be checked partially at compile time since the complex mechanism of Common Lisp parameter lists (e.g. `&key` parameters) allow the definition of functions with a varying number of arguments. These functions still need a runtime check.

The other implemented analysis method infers information about the number of values a form returns. This kind of information is called *multiple value information*. It is used in the code generation pass to generate code that handles multiple values efficiently.

To determine how many values a function may return, all forms in the body of the function which eventually produce the result, have to be analysed. These can be variables, special forms (in this case the analysis has to step through the subforms), or function calls.

The analysis method is implemented as a two stage algorithm. In the first stage for each function we associate the multiple value information infered by some rules describing the behaviour of LISP forms w.r.t. the passing of multiple values. Together with that information each function f is associated with all functions called in the body of f which yield the result values of f.

The second stage of the algorithm solves the dependencies of the multiple value informations using a fixpoint iteration.

`Example:`

```
(defun even-fac (acc n)
   (if (zerop n) acc (odd-fac (* acc n) (1- n))))

(defun odd-fac (acc n)
   (even-fac (* acc n) (1- n)))
```

Figure 1:

The function definitions of figure 1 leads to the following multiple value information in the first stage of the algorithm:

`even-fac:`	returns one value (`acc`) and calls `odd-fac`.
`odd-fac :`	calls `even-fac`.

The second stage of the algorithm resolves this system of dependencies and as the final result it yields the information that both functions deliver exactly one value.

3.3 Code Generation

The target language for CLICC is C as it is described in [KR83]. We do not use ANSI-C, because today it is not widely used as a standard. C serves as a portable high level

machine language. There is no direct correspondence between the language constructs of CL_1 and those of C. Most of the constructs of CL_1 have to be compiled into simpler constructs in C. Common $Lisp_1$ functions are compiled into C functions which play the role of low level subroutines. In C there are only global functions using the same global environment. The function concept of CL_1 includes nested function definitions, functional arguments and functional results (closures). Common $Lisp_1$ has a complex parameter passing mechanism. The order of evaluation of the parameters is specified as 'left to right'. In general, variables are statically scoped but they can be declared as dynamically scoped. We have to manage free local variables of local functions. Local Variables are allocated in the heap, if they occur free in closures, otherwise they are allocated in the LISP stack. We don't use the C runtime stack but a seperate LISP stack for parameter passing and the allocation of local variables. This allows us to determine the order of evaluation of parameters and the garbage collector has access to local variables when it needs to find all the references to data objects in the heap. The language constructs of CL_1 for non local control transfer allow non local function exits. They are implemented using the `setjmp` and `longjmp` constructs of C.

An Example

The following example shows a very simple CL_1 *program* and the generated C source code.

```
(in-package "USER" :use '("LISP"))

(defun list-copy (x)
  (cond
    ((null x) nil)
    (T (cons (car x) (list-copy (cdr x)))) ))

(print (list-copy (list 1 *x* "string")))
```

The generated code consists of two C functions. The first is the compilation of `list-copy`, the second is the compilation of the main part of the original CL_1 program, which consists of the code for `(in-package "USER" :use '("LISP"))` and `(print (list-copy (list 1 *x* "string")` in our example. The header files `<c_decl.h>` and `<fun_decl.h>` contain type definitions, macro definitions and function declarations used by the generated code. The included file `"list-copy.h"` is described below.

The C `main` function is part of the CLICC runtime library. It initializes the runtime system and calles the function `Flisp_main`.

```
#include <c_decl.h>
#include <fun_decl.h>
#include "list-copy.h"

U1_LIST_COPY( base )
CL_FORM *base;
{
   if( !(base[0].tag != CL_NIL) )
   {
      base[0].tag = CL_NIL;
      base[0].val.form = (CL_FORM *)&symbols[0];
   }
```

```
   else
   {
      base[1] = base[0];
      Fcar( &base[1] );
      base[2] = base[0];
      Fcdr( &base[2] );
      U1_LIST_COPY( &base[2]);
      Fcons( &base[1] );
      base[0] = base[1];
   }
}

Flisp_main(base)
CL_FORM *base;
{
   base[0].tag = CL_SMSTR;
   base[0].val.form = (CL_FORM *)&const_forms[0];
   base[1].tag = CL_SYMBOL;
   base[1].val.form = (CL_FORM *)&symbols[SYMSIZE * 69];
   base[2].tag = CL_CONS;
   base[2].val.form = (CL_FORM *)&const_forms[4];
   Fin_package( &base[0], 3 );
   base[0].tag = CL_FIXNUM;
   base[0].val.i = 1;
   base[1] =  (CL_FORM *)&symbols[SYM_SIZE * 223 + OFF_SYM_VALUE];
   base[2].tag = CL_SMSTR;
   base[2].val.form = (CL_FORM *)&const_forms[6]);
   Flist( &base[0], 3);
   U1_LIST_COPY( &base[0] );
   Fprint(&base[0], 1);
}
```

LISP functions are translated into C functions. The identifiers of LISP functions cannot be transformed directly into C identifiers. The identifier `list-copy` gets the prefix `U1_` (first user defined function). This prevents name conflicts even if the C compiler recognizes only the first few characters of an identifier.

We have to introduce a seperate LISP stack for parameter passing and local variables. Every translated LISP function gets a parameter `base`, which contains a pointer to the start of the activation record of the function. Activation records are subsequently allocated in the LISP stack if a function is called and they are deallocated if the execution of the function terminates. The result of a function call is stored into the first entry of the activation record.

The number of actual parameters is checked at compile time, if possible. Functions which allow a variable number of parameters get an additional C parameter, which describes the number of actual parameters. See for instance the function calls of `Flist` and `Fprint` in `Flisp_main`.

The constants `"USER"`, `'("LISP")` and `"string"` are translated into initialisation code for the C array `const_forms`. The symbols, i.e `NIL`, and global variables, i.e. `*x*`, are translated into the initialisation code for the C array `symbols`. The definitions of these arrays are put into the file `"list-copy.h"` which is not shown here. The array `symbols` contains all of the

symbols defined by the compiled program and the runtime system. The `in-package` top level form is compiled into the first seven lines of `Flisp_main`.

LISP data objects are translated into C data objects of the type `CL_FORM`, which contain value and type information. Thus, for example, the assignment of `NIL` to the result position of `U1_LIST_COPY` in the first `if` clause is performed by assigning `CL_NIL` to the `tag` field and the appropriate value to the `val.form` field.

We rely on the optimizations performed by the C compiler. It should for instance remove the duplicate negation in the condition part of the `if` statement in function `U1_LIST_COPY`.

4 Generating Executable Applications

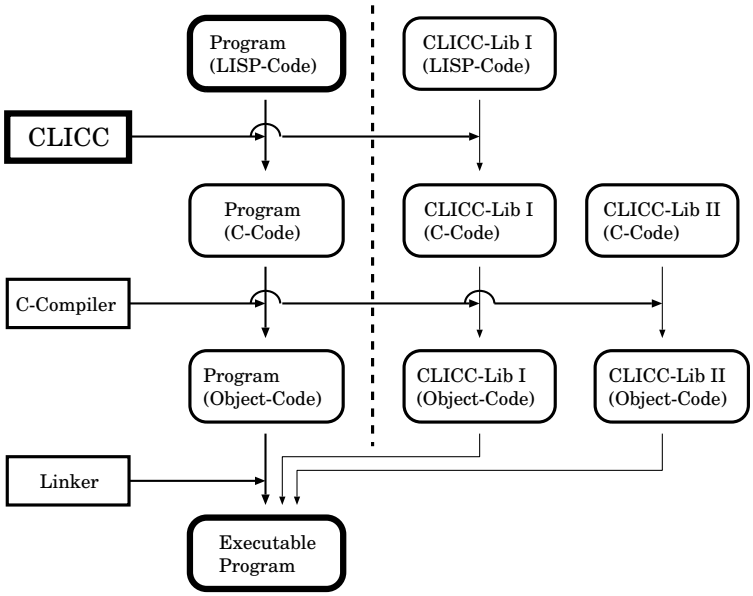

Figure 2:

Executable applications are generated in three steps. CLICC translates a CL_1 source file into a C file as described in figure 2. This file is translated by the C compiler to the machine language of the target machine and we get an object file. This file is not executable, because there are unresolved references to system functions of the LISP and C runtime library. These references are resolved by the linker, which generates an executable file. It contains all the referenced functions of the runtime libraries.

The LISP system functions are handcode in C or implemented in CL_1. The basic functions are written in C (e.g. `CAR, CDR, WRITE-CHAR`, Garbage Collector). More complex functions are written in LISP (e.g. `PRINT, READ`, package functions, most of the sequence functions). We use CLICC to translate those functions to C. The system functions are collected in two library files, which are used by the linker to generate the executable file.

Now let us take a closer look to the linking process. The generated executable files should be small, compared with the size of LISP-systems or saved images of LISP-programs which are loaded into LISP-systems. We use the linking mechanism of C to generate executable files of reasonable size.

If a C program is linked, all needed system functions of the C runtime library are included into the generated executable file. The system functions which do not occur in the translated

C program are not included.

We want to achieve this behaviour for LISP system functions, too. They are seperately compiled and put into an archive file, the CLICC runtime library. See figure 3. In full Common Lisp it is unknown at link time, which functions are to be included into the executable file because symbols may be transformed into functions at runtime. This is impossible in CL_1. Thus, we do know all the names of those system functions we have to link into the executable file and we are allowed to use the linker of the target system to perform this job.

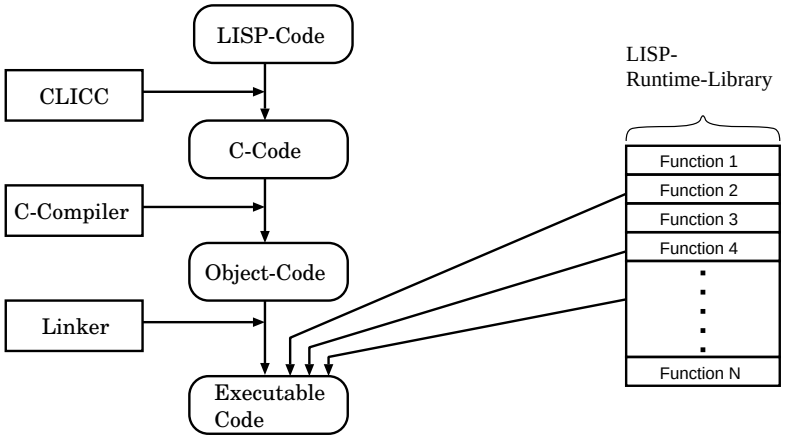

Figure 3:

The CL_1-coded system functions often call many other system functions. Thus, a large number of system functions is linked into the executable files even if the source file only uses a few of them.

5 Portability and Compiler Bootstrapping

Since CLICC is written in CL_1, it may be compiled by itself. CL_1 is a strict subset of Common Lisp and therefore we are able to run CLICC within a usual Common Lisp system to compile itself to C. Whether or not CLICC is able to reproduce itself is known to be a good test for a compiling algorithm, usually called the *bootstrap test*. It is shown in the figure 4.

We performed this test on a SUN SPARC1 workstation, compiling CLICC to C using the original CL_1-program within Lucid Common Lisp to produce the first CLICC in C, and it's output compiled with the system's C compiler to generate the second. Both of the former C programs were successfully compared with the UNIX `diff` command to be syntactically equal.

The portability of CL_1 programs on the basis of the generated C code is demonstrated in figure 5 where it is shown for CLICC, which, of course, is only one example program. Our approach yields the portablility for every *application program* written in CL_1.

6 Conclusions

We have defined an *efficiently implementable* strict subset Common $Lisp_1$ of Common Lisp, and a compiler CLICC from this subset to C source code.

The compilation is divided into several passes producing different intermediate program representations, i.e. Common $Lisp_0$, the abstract representation of Common $Lisp_0$ with anno-

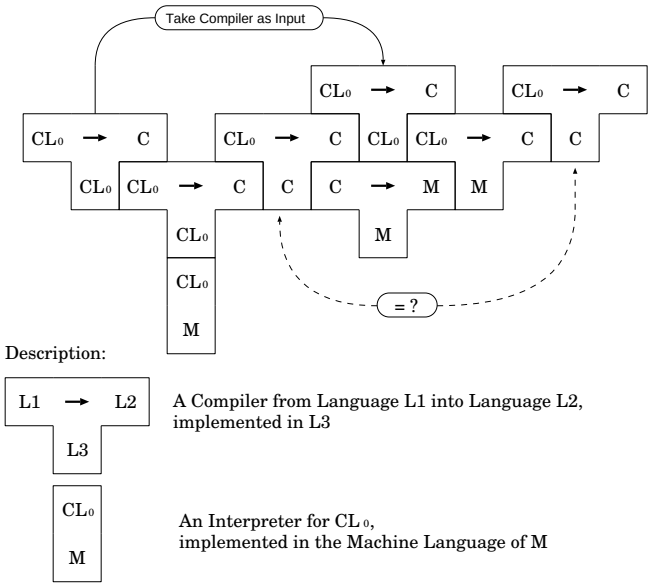

Figure 4:

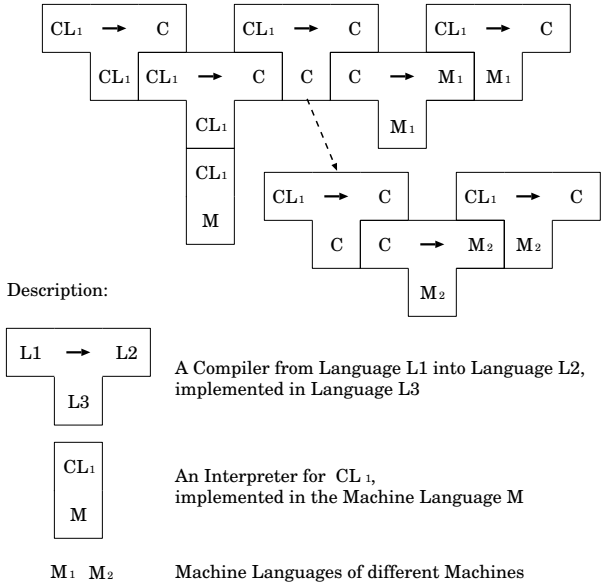

Figure 5:

tations produced by the global program analysis, and C code. Common Lisp_0 is a "minimal" kernal language to implement Common Lisp_1.

Further work has to be done on global analysis and optimization of programs. Type inference would be extremely useful not only to optimize user programs but also to increase the efficiency of the LISP-coded runtime system functions. Moreover, the compilation of LISP standard function calls into less generic runtime system function calls will reduce the code length of compiled and linked executable applications.

C is used as a machine oriented and portable target language. Although, of course, machine code generation for special machine architectures *can* produce more efficient code than portable C code ever can be, the price we actually have to pay for the sake of portablility is not yet clear. Of course, C code generation directly benefits from C compiler improvements.

7 References

[AG87] D. Ackermann and W. Goerigk. Das Projekt: Übersetzung von Wissensbasen. In *Workshop der Fachgruppe 2.1.4. „Alternative Konzepte für Sprachen und Rechner" der Gesellschaft für Informatik*, Bad Honnef, 1987.

[AGS88] D. Ackermann, W. Goerigk, and F. Simon. Kompilation von Wissensrepräsentationssprachen am Beispiel von BABYLON. Institutsbericht Nr. 8810, Institut für Informatik und Prakt. Math. der Christian-Albrechts Universität, Kiel, 1988. Also published as chapter 8 in [CdV89].

[Bur91] 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.

[CdV89] Th. Christaller, F. diPrimio, and A. Voß, editors. *Die KI–Werkbank BABYLON.* Addison–Wesley, Bonn, 1989.

[Knu91] 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.

[KR83] B.W. Kernighan and D.M. Ritchie. *Programmieren in C.* Hanser Verlag, München, Wien, 1983.

[MAE+62] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.I. Levin. *LISP 1.5 Programmer's Manual.* The MIT Press, Cambridge, MA, 1962.

[McC60] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. In *Communications of the ACM*, number 4 in Vol. 3, 1960.

[Ste84] G.L. Steele. *Common Lisp: The Language.* Digital Press, Burlington, 1984.

[Ste90] G.L. Steele. *Common Lisp: The Language. Second Edition.* Digital Press, Bedford, MA, 1990.

[TM86] Y. Taiichi and H. Masami. Kyoto Common Lisp Report. Technical report, Research Institute for Mathematical Science, Kyoto University, 1986.

A The Syntax of Common Lisp$_1$

The Syntax of Common Lisp$_1$ programs is defined by the following grammar:

cl$_1$-program ::= { *top-level-form* }*

top-level-form ::=
    (`IN-PACKAGE` *package-name* `&key :nicknames :use`) |
    (`PROVIDE` *module-name*) |
    (`REQUIRE` *module-name*) |
    (`LOAD` *file-name*) |
    (`PROCLAIM` *decl-spec-form*) |
    (`DEFVAR` *name* [*initial-value* [*documentation*]]) |
    (`DEFPARAMETER` *name initial-value* [*documentation*]) |
    (`DEFCONSTANT` *name initial-value* [*documentation*]) |
    (`DEFTYPE` *name bq-lambda-list* [*doc-string*] *bq-form*) |
    (`DEFSETF` *access-fn* { *update-fn* [*doc-string*] |
                        bq-lambda-list (*store-variable*)
                        [*doc-string*] *bq-form* }) |
    (`DEFSTRUCT` *name-and-options* [*doc-string*] { *slot-description* }*) |
    (`DEFMACRO` *name bq-lambda-list* [*doc-string*] *bq-form*) |
    (`DEFUN` *name lambda-list* [[{ *declaration* }* | *doc-string*]] { *form* }*) |
    (`PROGN` { *top-level-form* }*) |
    form

form ::= *atom* | *call*
atom ::= *self-evaluating-form* | *symbol*
call ::= *special-form* | *macro-call* | *fn-call*

bq-form ::= *atom* | (`QUOTE` *s-expr*) | *bq-call*
bq-call ::= (`CONS` *bq-form bq-form*) |
        (`LIST` { *bq-form* }*) |
        (`LIST*` { *bq-form* }*) |
        (`APPEND` { *bq-form* }*) |

special-form ::=
    (`QUOTE` *s-expr*) |
    (`FUNCTION` *fn*) |
    (`SETQ` *var form*) |
    (`PROGN` { *form* }*) |
    (`LET` ({ *var* | (*var* [*value*]) }*) { *declaration* }* { *form* }*) |
    (`LET*` ({ *var* | (*var* [*value*]) }*) { *declaration* }* { *form* }*) |
    (`PROGV` *symbols values* { *form* }*) |
    (`LABELS` ({ (*name lambda-list* [[{ *declaration* }* | *doc-string*]] { *form* }*) }*)
        { *declaration* }* { *form* }*) |
    (`FLET` ({ (*name lambda-list* [[{ *declaration* }* | *doc-string*]] { *form* }*) }*)
        { *declaration* }* { *form* }*) |
    (`IF` *test then else*) |
    (`BLOCK` *name* { *form* }*) |
    (`RETURN-FROM` *name result*) |
    (`TAGBODY` { *tag* | *statement* }*) |

(GO *tag*) |
(MULTIPLE-VALUE-CALL *function* { *form* }*) |
(MULTIPLE-VALUE-PROG1 *form* { *form* }*) |
(CATCH *tag-form* { *form* }*) |
(UNWIND-PROTECT *protected-form* { *cleanup-form* }*) |
(THROW *tag-form result*) |
(THE *value-type form*)

macro-call ::= (*name* { *s-expr* }*)
fn-call ::= (*fn* { *form* }*)

fn ::= *symbol* | *lambda-expr*

lambda-expr ::= (LAMBDA *lambda-list* [[{ *declaration* }* | *doc-string*]] { *form* }*)

lambda-list ::= ({ *var* }*
[&optional { *var* | (*var* [*initform* [*svar*]]) }*]
[&rest *var*]
[&key { *var* | ({*var* | (*keyword var*)} [*initform* [*svar*]]) }*
[&allow-other-keys]]
[&aux { *var* | (*var* [*initform*]) }*])

bq-lambda-list ::= ({ *var* }*
[&optional { *var* | (*var* [*bq-form* [*svar*]]) }*]
[&rest *var*]
[&key { *var* | ({*var* | (*keyword var*)} }*
[&allow-other-keys]]
[&aux { *var* | (*var* [*bq-form*]) }*])

declaration ::= (DECLARE { *decl-spec* }*)
decl-spec-form ::= (QUOTE *decl-spec*)
decl-spec ::= (SPECIAL { *var* }*) | (INLINE { *name* }*) | (NOT-INLINE { *name* }*) |
(IGNORE { *var* }*)

name-and-options ::= *name* | (*name* { *struct-option* }*)
struct-option ::= :CONC-NAME | (:CONC-NAME {NIL | *generalized-string*}) |
:CONSTRUCTOR | (:CONSTRUCTOR *symbol*) |
:PREDICATE | (:PREDICATE *symbol*) |
:COPIER | (:COPIER *symbol*) |
:INCLUDE | (:INCLUDE *name* { *slot-description* }*) |
:PRINT-FUNCTION | (:PRINT-FUNCTION *fn*)
slot-description ::= *name* | (*name* [*initial-value* { *slot-option* }*])
slot-option ::= (:TYPE *type*) | (:READ-ONLY *form*)

s-expr ::= *atom* | (*s-expr* . *s-expr*)

doc-string, *documentation*, *file-name* ::= *string*

package-name, *module-name* ::= *generalized-string*

name, *var*, *svar*, *access-fn*, *update-fn*, *store-variable* ::= *symbol*

$$\left.\begin{array}{l} initial\text{-}value, initform, value \\ symbols, values \\ test, then, else \\ result \\ function, tag\text{-}form \\ protected\text{-}form, cleanup\text{-}form \end{array}\right\} ::= form$$

statement ::= *call*

generalized-string ::= (QUOTE *symbol*) | *string* | *character*

value-type and *type* are Common Lisp datatypes.

self-evaluating-form is a LISP value, which is neither a symbol nor a list.

The following non-terminals represent data objects of the specified datatype:

symbol ∈ SYMBOL,
keyword ∈ KEYWORD,
character ∈ CHARACTER,
string ∈ STRING and
tag ∈ INTEGER ∪ SYMBOL.

B Expansion of CL_1 System Macros

The expansion of the predefined system macros is decribed in terms of source to source code transformations.

```
(AND)                          ≡ T
(AND form)                     ≡ form
(AND form . more-forms)        ≡ (IF form (AND . more-forms) NIL)

(OR)                           ≡ NIL
(OR form)                      ≡ form
(OR atom . more-forms)         ≡ (IF atom atom (OR . more-forms))
(OR call . more-forms)         ≡ (LET ((newsym call))
                                      (IF newsym newsym (OR . more-forms))

(PSETQ)                        ≡ NIL
(PSETQ var form)               ≡ (PROGN (SETQ var form) NIL)
(PSETQ var form                ≡ (PROGN
    . more-var-form-pairs)             (SETQ var
                                           (PROG1 form
                                                  (PSETQ . more-var-form-pairs))
                                       NIL)

(PROG1 first                   ≡ (LET ((newsym first))
    form1 ... formn)                   form1 ... formn newsym)

(PROG2 first second            ≡ (PROGN first
    . more-forms)                      (PROG1 second . more-forms))

(WHEN test . more-forms)       ≡ (IF test (PROGN . more-forms) NIL)
```

```
(UNLESS test . more-forms)    ≡ (IF (NOT test) (PROGN . more-forms) NIL)

(COND)                        ≡ NIL
(COND                         ≡ (IF antecedent
    (antecedent . more-forms)       (PROGN . more-forms)
    . more-clauses)                 (COND . more-clauses))
(COND (antecedent)            ≡ (OR antecedent
    . more-clauses)                 (COND . more-clauses))
(COND (antecedent))           ≡ (VALUES antecedent)

(CASE atom                    ≡ (COND
    (key1 . forms1)                 (test1 . forms1)
    ...                             ...
    (keyn . formsn)                 (testn . formsn)
    where testi = T,                      if keyi = T or
                                             keyi = OTHERWISE
          testi = (OR),                   if keyi = ()
          testi = (EQL keyi atom),        if keyi atomic
          testi = (OR (EQL keyi1 atom)
                      ...
                      (EQL keyim atom)), if keyi = (keyi1 ... keyim)
(CASE call                    ≡ (LET ((newsym call))
    . clauses)                      (CASE newsym
                                        . clauses)

(TYPECASE atom                ≡ (COND
    (type1 . forms1)                (test1 . forms1)
    ...                             ...
    (typen . formsn)                (testn . formsn)
    where testi = T,                      if keyi = T or
                                             keyi = OTHERWISE
          testi = (TYPEP atom (QUOTE keyi)), otherwise
(TYPECASE call                ≡ (LET ((newsym call))
    . clauses)                      (TYPECASE newsym
                                        . clauses)

(RETURN)                      ≡ (RETURN-FROM NIL NIL)
(RETURN result)               ≡ (RETURN-FROM NIL result)

(LOOP . forms)                ≡ (BLOCK NIL
                                    (TAGBODY newsym
                                             (PROGN . forms)
                                             (GO newsym)))

(DO ({var [init [step]]}*)    ≡ (PROG ({var [init]}*)
    (end-test . result-forms)       {declaration}*
    {declaration}*                  newsym
    {tag | statement}*              (WHEN end-test
)                                       (RETURN (PROGN . result-forms))
```

```
                                    {tag | statement}*
                                    (PSETQ {[var step]}*)
                                    (GO newsym))

(DO* ({var [init [step]]}*)     ≡ (PROG* ({var [init]}*)
     (end-test . result-forms)        {declaration}*
     {declaration}*                   newsym
     {tag | statement}*               (WHEN end-test
)                                         (RETURN (PROGN . result-forms))
                                      {tag | statement}*
                                      (SETQ {[var step]}*)
                                      (GO newsym))

(DOLIST (var listform           ≡ (DO (var
             [resultform])             (newsym (THE (LIST (VALUES listform)))
     {declaration}*                            (CDR newsym))
     {tag | statement}*               )
)                                     ((ATOM newsym)
                                         (SETQ var NIL) [resultform])
                                      {declaration}*
                                      (SETQ var (CAR newsym))
                                      {tag | statement}*)

(DOTIMES (var countform         ≡ (DO ((newsym countform)
              [resultform])             (var 0 (1+ var))
     {declaration}*                   )
     {tag | statement}*               ((>= var newsym) [resultform])
)                                     (DECLARE (FIXNUM var newsym))
                                      {declaration}*
                                      {tag | statement}*)

(PROG {(var [init])}*           ≡ (BLOCK NIL
     {declaration}*                   (LET ({(var [init])}*)
     {tag | statement}*                   {declaration}*
)                                         (TAGBODY {tag | statement}*)))

(PROG* {(var [init])}*          ≡ (BLOCK NIL
     {declaration}*                   (LET* ({(var [init])}*)
     {tag | statement}*                   {declaration}*
)                                         (TAGBODY {tag | statement}*)))

(MULTIPLE-VALUE-LIST form)      ≡ (MULTIPLE-VALUE-CALL #'LIST form)

(MULTIPLE-VALUE-BIND            ≡ (MULTIPLE-VALUE-CALL
        ({var}*) values-form         #'(LAMBDA (&OPTIONAL {var}*
     {declaration}*                              &REST newsym)
     {form}*                                     (DECLARE (IGNORE newsym))
)                                                {declaration}*
                                                 {form}*)
                                    values-form)
```

```
(MULTIPLE-VALUE-SETQ          ≡ (MULTIPLE-VALUE-BIND (newsym_1 ... newsym_n)
    (var_1 ... var_n) form)           form
                                  (SETQ var_1 newsym_1 ... var_n newsym_n)
                                  newsym_1)

(LOCALLY {declaration}*       ≡ ((LAMBDA () {declaration}* {form}*))
    {form}*)

(DO-SYMBOLS                   ≡ (BLOCK NIL
    (var [package [result]])      (rt::DO-SYMBOLS-ITERATOR
    {declaration}*                    #'(LAMBDA (var)
    {tag | statement}*                    {declaration}*
)                                         (TAGBODY {tag | statement}*))
                                      [package])
                                  (LET ((var NIL))
                                      {declaration}* [result]))

(DO-EXTERNAL-SYMBOLS          ≡ (BLOCK NIL
    (var [package [result]])      (rt::DO-EXTERNAL-SYMBOLS-ITERATOR
    {declaration}*                    #'(LAMBDA (var)
    {tag | statement}*                    {declaration}*
)                                         (TAGBODY {tag | statement}*))
                                      [package])
                                  (LET ((var NIL))
                                      {declaration}* [result]))

(DO-ALL-SYMBOLS               ≡ (BLOCK NIL
    (var [result])                (rt::DO-ALL-SYMBOLS-ITERATOR
    {declaration}*                    #'(LAMBDA (var)
    {tag | statement}*                    {declaration}*
)                                         (TAGBODY {tag | statement}*)))
                                  (LET ((var NIL))
                                      {declaration}* [result]))

(WITH-OPEN-STREAM             ≡ (LET ((var stream))
    (var stream)                  {declaration}*
    {declaration}*                (UNWIND-PROTECT
    {form}*                           (PROGN {form}*)
)                                     (CLOSE var)))

(WITH-INPUT-FROM-STRING       ≡ (WITH-OPEN-STREAM
    (var string                   (var (MAKE-STRING-INPUT-STREAM
      &key index (start 0) end)                                  string start end))
    {declaration}*                {declaration}*
    {form}*                       (MULTIPLE-VALUE-PROG1
)                                     (PROGN {form}*)
                                      [(SETF index (FILE-POSITION var))]))

(WITH-OUTPUT-TO-STRING        ≡ (WITH-OPEN-STREAM
    (var [string])                (var (MAKE-STRING-OUTPUT-STREAM [string])
    {declaration}*                {declaration}*
```

```
    {form}*                          {form}*
)                                    [(GET-OUTPUT-STREAM-STRING var)])

(WITH-OPEN-FILE                  ≡ (LET ((stream (OPEN filename {option}*))
    (stream filename {option}*)        (newsym T))
    {declaration}*                   {declaration}*
    {form}*                          (UNWIND-PROTECT
)                                        (MULTIPLE-VALUE-PROG1
                                             (PROGN {form}*)
                                             (SETQ newsym NIL))
                                         (WHEN (STREAMP stream)
                                             (CLOSE stream :ABORT newsym))))
```

The following expansion rules are used for the macros which allow the access to generalized variables, where

place ≡ (*access-fn* arg_1 ... arg_n),
access-form ≡ (*access-fn* tmp_1 ... tmp_n), and
store-form ≡ (SETF *access-form store-var*),

or

place ≡ *var*,
access-form ≡ *var*, and
store-form ≡ (SETQ *var store-var*).

```
(PSETF place1 newvalue1          ≡ (LET* ( (tmp11 arg11) ... (tmp1m1 arg1m1)
       ...                               (store-var1 newvalue1)
       placen newvaluen)                 ...
                                         (tmpn1 argn1) ... (tmpnmn argnmn)
                                         (store-varn newvaluen)
                                       )
                                   store-form1 ... store-formn NIL)

(SHIFT place1 ... placen         ≡ (LET* ( (tmp11 arg11) ... (tmp1m1 arg1m1)
       newvalue)                         (tmp-shifted-out access-form1)
                                         (tmp21 arg21) ... (tmp2m2 arg2m2)
                                         (store-var1 access-form2))
                                         ...
                                         (tmpn1 argn1) ... (tmpnmn argnmn)
                                         (store-varn newvalue)
                                       )
                                   store-form1 ... store-formn
                                   tmp-shifted-out)

(ROTATEF)                        ≡ NIL
(ROTATEF place)                  ≡ (PROGN place NIL)
(ROTATEF place1 ... placen)      ≡ (LET* ( (tmp11 arg11) ... (tmp1m1 arg1m1)
                                         (store-varn access-form1)
                                         (tmp21 arg21) ... (tmp2m2 arg2m2)
                                         (store-var1 access-form2))
```

```
                                    ...
                                    (tmp_n1 arg_n1) ... (tmp_nm_n arg_nm_n)
                                    (store-var_n-1 access-form_n)
                                  )
                                store-form_1 ... store-form_n NIL)

(INCF place)                    ≡ (INCF place 1)
(INCF place delta)              ≡ (LET* ( (tmp_1 arg_1) ... (tmp_n arg_n)
                                    (store-var (+ access-form delta)
                                  )
                                store-form)

(DECF place)                    ≡ (DECF place 1)
(DECF place delta)              ≡ (LET* ( (tmp_1 arg_1) ... (tmp_n arg_n)
                                    (store-var (- access-form delta)
                                  )
                                store-form)

(PUSH item var)                 ≡ (SETQ var (CONS item var))

(PUSH                           ≡ (LET* ( (tmp_1 arg_1) ... (tmp_n arg_n)
    item                                  (store-var (CONS item access-form))
    (access-fn arg_1 ... arg_n))          )
                                    store-form)

(PUSHNEW item var               ≡ (SETQ var
    . key-value-pairs)                    (ADJOIN item var . key-value-pairs))

(PUSHNEW                        ≡ (LET* ( (tmp_1 arg_1) ... (tmp_n arg_n)
    item                                  (store-var (ADJOIN item access-form
    (access-fn arg_1 ... arg_n)                              . key-value-pairs))
    . key-value-pairs)                    )
                                    store-form)

(POP var)                       ≡ (PROG1 (FIRST var)
                                         (SETQ var (REST var)))

(POP (access-fn arg_1 ... arg_n)) ≡ (LET* ( (tmp_1 arg_1) ... (tmp_n arg_n)
                                      (store-var access-form)
                                    )
                                    (PROG1 (FIRST store-var)
                                           (SETQ store-var (REST store-var))
                                           store-form))
```