The
Connection Machine
System

*Lisp Compiler Guide

Version 5.0
September 1988

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, August 1986
Revised, September 1987
Revised, September 1988

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1214
(617) 876–1111

Contents

Preface

Objectives of This Manual

The **Lisp Compiler Guide* explains how to use the *Lisp Compiler and its options to maximize the performance of *Lisp programs.

Intended Audience

This guide is intended for programmers well versed in Common Lisp and comfortable using a Lisp environment. Some familiarity with the Common Lisp compiler and some familiarity with *Lisp are also assumed. The reader may be using either the Symbolics Lisp machine front end, Lucid Common Lisp on a VAX system running ULTRIX, or Lucid Common Lisp on a Sun workstation under UNIX.

Revision Information

The **Lisp Compiler Guide*, Version 5.0 supercedes all previous descriptions of the *Lisp Compiler. This is an updated and expanded version of the **Lisp Compiler Guide*, Version 4.2A Field Test.

Organization of This Manual

The **Lisp Compiler Guide* is divided into two parts. Part I, "*Lisp Compiler Features," describes the facilities made available with the *Lisp compiler. Part II, "*Lisp Compiler Practicum," provides a narrative tutorial detailing effective techniques writing *Lisp code that the *Lisp compiler can compile.

Part I. *Lisp Compiler Features consists of the following five sections.

1 Introduction
The *Lisp compiler is described as an extension to the Common Lisp compiler.

2 Running the Compiler

This section describes how to invoke the *Lisp Compiler. It then explains which *Lisp expressions the compiler will treat and how to display the code it generates.

3 Maximizing Performance

This section details how to write *Lisp code that, when compiled, will yield optimal perfomance. Emphasis is placed on how to write correct type declarations and on how to obtain performance gains by judicious use of a low safety compiler option.

4 Setting Compiler Options

This section explains two ways to set the compiler options: by using the *Lisp compiler options menu, and by setting the compiler variables directly.

5 Compiler Options Reference

This section lists each *Lisp compiler option with its possible values, default value, and variable. The different effects obtained by each possible value are explained.

Part II. *Lisp Compiler Practicum consists of the following 4 sections.

6 Tricks of the Trade

This section introduces the *Lisp compiler tutorial.

7 What Makes the *Lisp Compiler Work?

This section describes how the *Lisp compiler interacts with the Common Lisp compiler and explains what is required to ensure that *Lisp code is compiled. A simple uncompilable *Lisp function is changed to make it compilable.

8 How to Use Type Information

This section examines sample code and illustrates the proper use of **declare**, ***proclaim**, **the**, and ***locally** expressions.

9 How to Write General *Lisp Code that *Compiles

This section presents several techniques for generalizing *Lisp code to process a variety of data types while ensuring that it can be compiled by the *Lisp compiler.

Notation Conventions

Throughout this manual, we use the terms **compile*, **compilation*, and **compiled code* when referring to the process by which the *Lisp compiler translates *Lisp code into Lisp/Paris and the results of that process. This is done to make a clear distinction between normal Common Lisp compilation and the *Lisp compilation.

Names of language elements within text appear in a sans serif, boldface type, as in ***max**. User input appears in the same bold type.

Code examples are set in typewriter style typeface, as in:

```
(cons a b)
```

Metavariables, names that stand for pieces of code, appear in italics, as in:

(macroexpand-1 *form*)

Where appropriate, code examples use pvar names that indicate type. For example, an unsigned pvar 8 bits long might be named `u8`, and a second such pvar in the same code example might be named `u8-2`. Similarly a signed pvar of length 16 might be named `s16`.

Related Materials

- *The *Lisp Reference Manual*, Version 5.0
  This reference manual describes the essential elements of the *Lisp language.
- *Supplement to the *Lisp Reference Manual*, Version 5.0
  This supplement expands and updates *The *Lisp Reference Manual.*
- *Connection Machine Front-End Subsystems*
  This volume describes the various front-end computers used with the Connection Machine system.
- *Paris Reference Manual*, Version 5.0
  Paris (for *parallel instruction set*) is the Connection Machine system's instruction set. The *Lisp compiler generates a combination of Lisp and Paris instructions.
- *Common Lisp: The Language*, by Guy L. Steele Jr.
  This book defines the de facto industry standard Common Lisp and describes the Common Lisp compiler options.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142-1214
Internet Electronic Mail:	customer-support@think.com
Usenet Electronic Mail:	harvard!think!customer-support
Telephone:	(617) 876-1111

For Symbolics users only:

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press CTRL-M to create a report. In the mail window that appears, the `To:` field should be addressed as follows:

```
To:  bug-connection-machine@think.com
```

Please supplement the automatic report with any further pertinent information.

Part I
*Lisp Compiler Features

1 Introduction

Compiled *Lisp runs more efficiently than interpreted *Lisp. By compiling *Lisp code that previously ran only interpreted, improvements in performance of 25% to 500% have been observed. The *Lisp Compiler often generates better code than that written by experienced, human Paris programmers.

An interpreted *Lisp program executes as a sequence of calls to a set of built-in *Lisp functions, collectively called the **Lisp interpreter.* These functions interpret their arguments and, depending on the arguments' types, execute appropriate Lisp forms and Paris instructions. The *Lisp compiler generates Lisp/Paris object code that in several respects is more efficient than that produced by the interpreter. For instance, compiled *Lisp code avoids the overhead of run-time type determination. Also, compiled code almost always uses less stack space than interpreted code uses. In addition, unlike the interpreter, the *Lisp compiler can generate specialized Paris instructions that combine more than one *Lisp operation into a single Paris call.

The *Lisp compiler is compatible with the Common Lisp compiler, which is described in *Common Lisp: The Language.* When enabled, the *Lisp compiler executes as part of the Common Lisp Compiler.

The compilation process effected by the *Lisp compiler is unlike that of most compilers. To avoid ambiguity, therefore, the terms **compile*, **compilation*, and **compiled code* are used when referring to the process by which *Lisp compiler translates *Lisp code into Lisp/Paris and the results of that process. This is done to make a clear distinction between normal Common Lisp compilation and *Lisp compilation.

2 Running the Compiler

The *Lisp compiler is enabled by default. Invoking the Common Lisp compiler on any *Lisp program automatically invokes the *Lisp compiler. To compile a *Lisp form, use the Common Lisp **compile** function. To compile all definitions within a file containing *Lisp code, use the Common Lisp **compile-file** function. These two functions are defined in *Common Lisp: The Language.*

If the front-end machine provides additional commands for compiling Common Lisp code, these may be used to compile *Lisp code. For instance, the Symbolics Lisp machine editor commands **Meta-x Compile Buffer**, **Meta-x Compile File**, **Meta-x Compile Region**, and **Meta-x Compile Changed-Definitions** automatically invoke both the Common Lisp and the *Lisp compilers.

2.1 What Does (and Does Not) Get Compiled

The *Lisp compiler can *compile most *Lisp statements. It does not, however, *compile all *Lisp expressions. The *Lisp compiler handles all the expressions it can and the *Lisp interpreter handles the rest.

There are two criteria that *Lisp expressions must meet to ensure that the *Lisp compiler will attempt to *compile them.

1. Expressions must be *visible* to the *Lisp compiler.

   The *Lisp compiler attends to *Lisp statements that appear textually within certain macros and temporary variable assignments only.

2. Expressions must use only variables for which correct type declarations have been made—and these declarations must be *visible* to the *Lisp compiler.

   The *Lisp compiler cannot *compile expressions containing undeclared pvars.

If a *Lisp expression meets these criteria, it will likely be *compiled. However, some *Lisp forms are never *compiled. (For a current list of uncompilable *Lisp forms, see the current *Lisp Release Notes.)

2.1.1 Macros

The *Lisp compiler is invoked when the Common Lisp compiler expands a macro call. Most *Lisp macros therefore compile.

The *Lisp compiler attempts to *compile the following *Lisp expressions in their entirety. Statements textually within the scope of these forms are considered *visible* to the *Lisp compiler.

***set**	***pset**	***setf**
pref	***sum**	***integer-length**
***or**	***and**	***xor**
***logior**	***logand**	***logxor**
***max**	***min**	

Also, the predicates for ***when**, ***unless**, ***if**, and ***cond** and the initial values for ***let** and ***let*** variables are compiled. The compiler does not compile the body of these forms.

See Part II, "*Lisp Compiler Practicum," for examples illustrating how to write code that the *Lisp compiler can compile.

2.1.2 Type Declarations

The *Lisp compiler attends to type declarations for both pvars and front-end variables. Expressions containing undeclared pvars or functions that return values of undeclared type are never compiled.

There are two types expressions that are never compiled, although they contain pvars of declared type.

1. Expressions containing general pvars—pvars declared to be of type **(pvar t)**—are not compiled unless they are type predicates.

2. Expressions containing general mutable pvars—pvars declared to be of type **(pvar *)** or left undeclared—are are never compiled.

See section 3, "Maximizing Performance," for discussions on how to use various type declarations and on how to write flexible code with strict type adherence.

Example

Most but not all the expressions in the following code can be *compiled.

```
(*proclaim '(type (pvar (unsigned-byte 8)) field))
(*let ((foo (*!! field field)))
   (declare (type (pvar (unsigned-byte 32)) foo))
   (*set foo (*!! foo foo))
   (+!! foo foo))
```

Here, both ***!!** pvar expressions are compiled. The first ***!!** expression is included in the initial value form of a ***let**. The second ***!!** expression lies within a ***set** form. Notice however that the **+!!** pvar expression is interpreted rather than compiled; it is not within any of the forms that the *Lisp compiler handles.

2.2 Displaying Code that the Compiler Generates

In the process of compilation, *Lisp forms are macro-expanded by the Common Lisp compiler. The *Lisp compiler is invoked as a part of this macro-expansion. Therefore, it is relatively easy to look at the code generated by the *Lisp compiler.

Make sure that the *Lisp compiler is enabled, as it is by default. If necessary, to enable the *Lisp compiler, type **(setq *compilep* t)** or use the *Lisp compiler options menu as described in section 4.1, below.

On the Symbolics Lisp machine front end, from within the editor, the commands **Macro Expand Expression (Control-Shift-m)** and **Macro Expand Expression All (Meta-Shift-m)** may be used to expand a *Lisp form and display the results.

On any front end the functions **macroexpand** and **macroexpand-1** may be called directly on forms. Thus,

```
(pprint (let ((*compiling* t))
          (macroexpand-1 *Lisp-form)))
```

displays the Lisp/Paris code generated by the *Lisp compiler for **Lisp-form*.

Notice that the *Lisp variable ***compiling*** must be bound, as in the **let** form above, otherwise the *Lisp compiler will not process **Lisp-form*. When the **macroexpand-1** form is executed, the *Lisp compiler checks the value of ***compilep***. If ***compilep*** is **t**, meaning the *Lisp compiler is enabled, the *Lisp compiler checks the variable ***compiling***. If ***compiling*** is also **t**, meaning that compilation is in progress, the expression is *compiled by the *Lisp compiler.

The following code illustrates the results of compiling a few lines of *Lisp code.

```
(*proclaim '(type (signed-pvar 16) s16 s16-2))
(*proclaim '(type (signed-pvar 8) s8 s8-2))
(*proclaim '(type boolean-pvar b1))
(*set s16 (+!! (*!! s8 s8-2) s16-2))
(*sum (if!! b1 s8 s8-2))
```

The ***set** expression above generates the Lisp/Paris code below. Notice that no stack space is used to compute the result of the ***set** expression; no operations manipulating the stack are present.

```
(progn
  (cm:multiply (pvar-location s16)
               (pvar-location s8)
               (pvar-location s8-2) 16 8 8)
  (cm:+ (pvar-location s16) (pvar-location s16-2) 16)
  (cmi::error-if-location cm:overflow-flag 66575)
  nil)
```

The ***sum** expression in the code example above generates the following Lisp/Paris code.

```
(let* ((slc::old-next-stack-field (cmi::next-stack-field))
       (if!!-context-1 (+ slc::old-next-stack-field 8))
       (if!!-index-2 (+ if!!-context-1 1)))
  (prog1
    (progn
      (cm:allocate-stack-field
        (- if!!-index-2 slc::old-next-stack-field))
      ;; Save context - if!!
      (cmi::store-context-always if!!-context-1)
      ;; Compute predicate and select processors
      ;; where the predicate is true - if!!.
      (cmi::load-context (pvar-location b1))
      ;; Compute and move then clause in context of
      ;; processors where the predicate is true - if!!.
      (cm:move slc::old-next-stack-field (pvar-location s8) 8)
      ;; Select processors where the predicate is false. - if!!.
      (cm:logandc1-always cm:context-flag if!!-context-1 1)
      ;; Compute and move else clause in context of
      ;; processors where predicate is false - if!!.
      (cm:move slc::old-next-stack-field (pvar-location s8-2) 8)
      ;; Restore context - if!!.
      (cmi::load-context-always if!!-context-1)
      (cm:global-add slc::old-next-stack-field 8))
    (cm:deallocate-upto-stack-field slc::old-next-stack-field)))
```

As the above example illustrates, the Lisp/Paris code generated by the *Lisp compiler includes internal, undocumented Paris functions. This is done to make the code generated by the compiler more efficient; it should not be construed as an invitation to use undocumented Paris functions. Nonetheless, if questions about the code generated by the *Lisp compiler arise, feel free to contact Thinking Machines Customer Support.

3 Maximizing Performance

3.1 Type Declarations

This section explains how to communicate type information to the *Lisp compiler so that all expressions that can potentially be compiled *are* compiled. For a general discussion of *Lisp pvar types, see the chapter entitled "Parallel Variable Types" in the *Supplement to the *Lisp Reference Manual.*

Expressions containing undeclared pvars are not compiled. To take full advantage of the performance gains possible with compiled code, it is essential to use type declarations. Here is a list of basic principles governing the interaction between type declarations and the *Lisp compiler.

1. A type declaration is a promise made by the programmer to the compiler. It guarantees that only values of the specified type will be assigned to a variable or form declared to be of that type.

2. Type declarations do not direct the compiler to produce type coercions.

3. Correct type declarations do not affect the semantics of a correct program.

4. It is an error for a program to violate a type declaration; the results of an incorrect declaration are unpredictable.

5. If a type declaration is changed, all code depending on that declaration must be recompiled.

6. Expressions containing undeclared pvars never compile. Undeclared pvars default to the general mutable type `(pvar *)`; these are ignored by the compiler.

7. Expression containing general pvars **(pvar t)** do not compilek. There is one exception: type predicates. A general pvar used within a type predicate form *is* recognized by the compiler.

Examples

The following examples illustrate the principles described above.

Principle 1. Declarations are promises.

The **the** form below advises the compiler that **catamaran** will always be a double-precision floating-point value.

```
(*defun biggest-float (catamaran)
  (*max (the (pvar double-float) catamaran)))
```

Principle 2. Declarations do not cause type coercion.

Given principle 1, it is the responsibility of the programmer, *not* of the compiler, to ensure that type declarations are not violated. If a value needs to be coerced into a variable's declared type before being bound, the function **coerce!!** should be explicitly called.

```
(*proclaim '(type (pvar single-float)) my-two-hull)
(*defvar my-two-hull)
(*set my-two-hull (!! 2.5))
(biggest-boat (coerce!! my-two-hull '(pvar double-float)))
```

Here, **my-two-hull** begins as a single-float pvar and is coerced into a double-float pvar and the result is passed to the **biggest-boat** function.

Principle 3. Correct declarations do not affect program semantics.

Adding a type declaration simply informs the compiler that a variable is guaranteed to be of a specific type; it does not make code more correct. For example, if both functions defined below are always called with unsigned-byte pvar arguments, they will operate in exactly the same manner at run time, but the second will be faster.

```
(*defun  in-range (foo bar)
  (*and (>!! bar (self-address!!) foo)))

(*defun in-range2 (foo bar)
  (declare (type (unsigned-byte-pvar *) foo bar))
  (*and (>!! bar (self-address!!) foo)))
```

What distinguishes **in-range2** from **in-range** is that **in-range2** declares its arguments. As a result of this declaration, **in-range-2** will compile into Lisp/Paris code. However, **in-range** is clearly more general; it has no type declarations and may be called with any numeric pvar.

Principle 4. Violating a type declaration leads to unpredictable results.

The function definition below declares its parameter, x , to be a 32-bit unsigned integer pvar.

```
(defun *sum-u32 (x)
  (*sum (the (field-pvar 32) x)))

(*sum-u32 (!! 1.0))            ;single-float, 32 bits
(*sum-u32 (!! 1))              ;unsigned 1 bit
(*sum-u32 (!! 1.0D0))          ;double-float, 64 bits
```

None of these three calls to ***sum-u32** are valid. The first call takes a 32-bit floating point representation in each processor and attempts to sum it as a 32-bit integer representation. The second call takes a bit containing the value 1 and 31 bits of random data from each processor and sums these meaningless values. The third call sums the low-order 32 bits of a double-precision representation of 1 and ignores the high-order 32 bits in each processor. These errors will *not* be signaled at a compiler **Safety** level of **0**; they will be signaled at **Safety** level **3**. (See section 3.3 for a discussion of safety levels.)

A valid call to ***sum-u32** must provide a correctly typed argument. For instance,

```
(*sum-u32 (coerce!! (!! 1) '(field-pvar 32)))
```

can be correctly used with the definition of ***sum-32** above. Also,

```
(*proclaim '(type (field-pvar 32) u32))
(*defvar u32)
(*set u32 (!! 1))
(*sum-u32 u32)
```

can be correctly used with the definition of ***sum-32** above.

Principle 5. If a type declaration is changed, related code must be recompiled.

Below, **inner-tube** is proclaimed to be a pvar containing single-precision floating-point numbers. The function **foo** uses this variable.

```
(*proclaim '(type (pvar single-float) inner-tube))
(*set inner-tube (!! 2.0))
(defun foo ()
  (*sum inner-tube))

;compile the above

(*proclaim '(type (pvar double-float) innner-tube))
(*defvar float-pvar)

(foo)      ;This does not work until the definition of foo
           ;has been recompiled with the new inner-tube
           ;proclamation.
```

The initial proclamation and the definition of **foo** are compiled. Then, **inner-tube** is proclaimed again, differently. The call to **foo** is therefore incorrect. Until **foo** is recompiled with the second proclamation, any call to it is incorrect.

Principle 6. Expressions containing undeclared or general mutable pvars are never compiled.

Below, **mut-pvar** is declared general mutable. The pvar **undeclared** is undeclared and therefore defaults to the general mutable type.

```
(*proclaim '(type (pvar *) mut-pvar))
(*defvar mut-pvar)
(*set mut-pvar (random!!))
(*defvar undeclared)
(*set undeclared (self-address!!))
```

Although ***set** forms are visible to by the compiler, the ***set** forms above can not be compiled because they contain general mutable pvars.

Principle 7. Expressions containing general pvars never compile—with the exception of type predicate forms.

```
(*and (integerp!! (the (pvar t) my-general)))

(*set (the (pvar t) general) (!! nil))
```

The first of the above forms can compile, the second cannot.

3.1.1 Pvar Types

The *Lisp compiler currently recognizes all *Lisp pvar types except general pvars **(pvar t)** and general mutable pvars **(pvar *)**. The compiler ignores general pvars except when used in type predicates. General mutable pvars are always ignored by the compiler.

All other classes of *Lisp pvar types are recognized. Theses are;

front-end	**boolean**
character	**string-chars**
unsigned-byte	**signed-byte**
defined-float	**complex**
array	**structure**

For each pvar type there are pseudonyms and alternative type specifiers. For a complete list, check the chapter entitled "Parallel Variable Types" in *Supplement to the *Lisp Reference Manual.*

The *Lisp compiler also recognizes mutable pvars of definite type. A pvar is declared mutable by specifying the length value or values as *. For instance, the following code declares mutable float, mutable complex, mutable unsigned-byte, and mutable signed-byte pvars. It will compile.

```
(*defun sum-float (float-mut)
  (declare (type (pvar (defined-float * *))) float-mut)
  (*sum float-mut))

(*defun sum-complex (complex-mut)
  (declare (type (pvar (complex (defined-float * *))))
               complex-mut)
  (*sum complex-mut))

(*defun sum-unsigned (unsigned-mut)
  (declare (type (pvar (unsigned-byte *))) unsigned-mut)
  (*sum signed-mut))

(*defun sum-signed (signed-mut)
  (declare (type (pvar (signed-byte *))) signed-mut)
  (*sum signed-mut))
```

In addition, types defined by the Common Lisp **deftype** macro are supported by the compiler. For example,

```
(deftype my-complex-pvar ()
  '(pvar (complex (defined-float 36 8))))
```

```
(*defun frib-complex (foo)
  (declare (type my-complex-pvar foo))
  (*and (and!! (<!! (realpart!! foo) (!! 2.0)))))

(*proclaim '(type my-complex-pvar bar))
(*defvar bar)
(frib-complex bar)
```

defines and uses a new type restricted to complex pvars with 36-bit significands and 8-bit exponents.

3.2 How to Use Various Type Declarations

To communicate type information to the *Lisp compiler, use the *Lisp ***proclaim** or ***locally** constructs or the Common Lisp **declare** or **the** constructs.

*Proclaim versus Proclaim

To make global pvar type declarations, ***proclaim** should be used instead of **proclaim**. The *Lisp function ***proclaim** is equivalent to the Common Lisp function **proclaim** with one crucial exception. *Lisp arranges to have ***proclaim** forms evaluated at compile time. In contrast, the *Lisp compiler sees a **proclaim** form at compile time only if it has been previously evaluated.

```
(proclaim '(type (pvar (complex single-float)) balloon!!))
(*proclaim '(type (pvar (complex single-float)) wind-surfer!!))
```

The **balloon!!** type differs from the **wind-surfer!!** type in that the *Lisp compiler may not receive the **balloon!!** type specification at compile time. ***Proclaim** forms are guaranteed to be seen by the *Lisp compiler.

NOTE

The use of **proclaim** forms in *Lisp is obsolete; neither the *Lisp interpreter nor the *Lisp compiler will attend to **proclaim** forms in future releases.

The following kind of proclamation forms can be used to globally declare variable and function types:

```
(*proclaim '(type ...))
(*proclaim '(ftype ...))
(*proclaim '(function ...))
```

The Common Lisp **type** proclamation is used to globally specify the type of a variable or pvar. The Common Lisp **ftype** and **function** proclamations are used to globally specify the return type of user-defined functions. For example:

```
(*proclaim '(ftype (function (t) boolean) willow-tree))
(*set b1 (!! (willow-tree 0.0)))
```

In the example above, a single type proclamation for the user-defined Lisp function **willow-tree** replaces multiple occurrences of type declarations such as this one:

```
(*set b1 (!! (the boolean (willow-tree 0.0))))
```

The Common Lisp **ftype** proclamation can be used both to declare that a user-defined *Lisp function returns a pvar and to specify the type of that pvar. For example, the following form declares that the function **surface-area!!** returns a single-float pvar:

```
(*proclaim '(ftype (function (t t) (pvar single-float))
                    surface-area!!))
```

The Common Lisp **function** proclamation may be used instead of **ftype**; the two forms are interchangeable. Thus, the form above can be composed as:

```
(*proclaim
  '(function surface-area!! (t t) (pvar single-float)))
```

Note that in the **function** and **ftype** examples above, **t** is given as the type specifier for function arguments—meaning that any type is allowed. More exact type specifiers may be used in the argument type list, but the *Lisp compiler does not use this information. The examples therefore use the simplest form.

Declare

Declarations using the **declare** special form must include the **type** form as in:

```
(declare (type (float-pvar 23 8) sail-boat-pvar))
```

The following declaration is incorrect.

```
(declare ((float-pvar 23 8) incorrect-pvar))
```

The compiler issues a warning when a **declare** form appears without enclosing a **type** form. In later releases, an error may be signaled at compile time if this kind of declaration is attempted.

The placement of **declare** forms determines whether or not the *Lisp compiler will attend to them. The compiler receives type information only from **declare** forms placed within ***defun**, ***locally**, ***let**, and ***let*** forms. The comments in the following example indicate which declarations the compiler receives.

```
(*defun bax (f8)
  ;; This declaration is received by the *Lisp compiler
  (declare (type (field-pvar 8) f8))
  (let ((a 5.0))
    ;; This declaration is not received by the *Lisp compiler,
    ;; so the (!! a), in the following *let is not compiled.
    ;; This declaration is however received by the Common Lisp
    ;; Compiler.
    (declare (type single-float a))
    (*let ((b (+!! f8 (!! a))))
       ;; This declaration is received by the *Lisp compiler.
       (declare (type (pvar single-float) b))
       b)))
```

In addition to communicating pvar type information, the **declare** form may be used within ***defun**, ***locally**, ***let**, and ***let*** forms to give function type information. For example,

```
(*defun oof ()
  (declare (ftype (function (t) single-float-pvar) my-fun!!))
  (*sum (my-fun!! (!! 1.0))))
```

Here, **my-fun!!** is declared to return a single-precision floating-point pvar.

The

The Common Lisp special form **the** may be used to declare that the value of an unnamed form is a pvar type. For example,

```
(the (pvar boolean) (is-it-p!! some-pvar))
```

guarantees that the **is-it-p!!** form will evaluate to a boolean pvar.

Always Declare the Argument Type in a Call to !!

The compiler must be supplied with the type of the argument to the !! function. For this purpose, scalar Common Lisp types such as fixnum, single-float, double-float, integer, unsigned-byte, signed-byte, bit, and boolean may be used. For example,

```
(*proclaim '(type float-pvar var))
(*set var (+!! var (!! (the single-float some-expression))))
```

is guaranteed to compile. In contrast,

```
(*proclaim '(type float-pvar var))
(*set var (+!! var (!! some-expression)))
```

will not compile.

*Locally

The new *Lisp ***locally** construct allows the specification of declarations that apply only within the scope of the form. Using this form avoids the repetitious use of **the** forms.

***locally** *declaration-1 declaration-2 ... declaration-n* **&body** *body* [*Macro*]

This macro is used to provide declarations for the *Lisp compiler. The declarations *declaration-1* through *declaration-n* are used by the compiler within *body*. A ***locally** declaration must be a **declare** form. Any valid compositions of **declare** may be used within a ***locally** form, including **optimize** and ***optimize** forms.

In previous releases, declarations would only be seen by the *Lisp compiler when used within a ***defun**, ***let**, or ***let*** form. With the use of ***locally**, the user is now able to give the *Lisp compiler type information and optimization directives anywhere in a program.

Examples:

```
(defun locally-test (j)
 (*locally
   (declare (type fixnum j))
   (*let (temp)
     (declare (type (unsigned-byte-pvar 32) temp))
     (*set temp (!! j)))))

(defun *locally-example (result)
(*locally
  (declare (type single-float-pvar result))
  (do-for-selected-processors (j)
    (*locally
      (declare (type fixnum j))
      (flet
        ((local-pvar-function (x)
           (*locally
             (declare (type single-float-pvar x result))
             (declare (*optimize (safety 0)))
             (*set result (+!! x (!! j))))))
        (dotimes (i *current-cm-configuration*)
          (*locally
            (declare (type fixnum i))
            (*let ((temp (*!! (+!! (float!! (!! i)) (!! j))
                                (sin!! (!! j)))))
              (declare (type single-float-pvar temp))
              (local-pvar-function temp))))))))
```

Without ***locally**, the *Lisp compiler could handle the expressions in the examples above that include **(!! j)** only if each use of **(!! j)** were rendered as **(!! (the fixnum j))**. In most cases, using ***locally** once within each enclosing form is easier and less prone to error.

Notice that ***locally** allows declaration of the arguments to local functions defined by **flet** and **labels**. Previously there was no way to do this.

Using Length Expressions in Type Specifiers

Length expressions in *Lisp type specifiers are evaluated by the *Lisp compiler. The more complicated the length expression within a pvar type declaration, the less efficient is the compiled code using that pvar. From most to least efficient, the following hierarchy holds for length specification: constant, symbol evaluating to constant, expression evaluating to constant, mutable.

```
(*proclaim '(type (unsigned-byte 8) magenta))
(*proclaim '(type (unsigned-byte bar) magenta))
(*proclaim '(type (unsigned-byte (+ foo bar)) magenta))
(*proclaim '(type (unsigned-byte *) magenta))
```

Alternate proclamations for the variable **magenta** are shown above in order of decreasingly efficient compiler output.

NOTE

When a length expression is used in a type specifier, the compiler generates a reference to it. The length expression should have no side effects because it may be evaluated many times.

3.3 Safety Optimization and Error Detection

*Lisp attempts to catch and signal all detectable user errors. This is important for rapid program development, but detecting errors increases program execution time. The **Safety** compiler option controls the kind of code that is generated for error detection and the way errors are reported. While debugging, the safety level should be set to high. When running debugged code, lower the safety level to decrease the time spent detecting errors and thereby reduce execution time.

In the following example, the **+!!** expression may produce a nine-bit result, which cannot fit in the eight-bit ***set** destination. If the result is too large, an integer overflow error occurs.

```
(proclaim '(type (field-pvar 8) u8 u8-2))
(*set u8 (+!! u8 u8-2))
```

The code produced by the compiler is:

```
(progn (cm:u+ (pvar-location u8) (pvar-location u8-2) 8)
       (error detection code)
       nil)
```

The error detection code produced depends on the safety level at compilation time.

Available *Lisp compiler safety levels are listed below, along with the error-detection code produced for the example above when each is in effect. (Note that the number 66575, which appears in some error detection code, is a tag encoding information such as the *Lisp function causing the error. This number bears no particular significance; it is used mearly for purposes of illustration.)

Safety 0

(No error detection code is produced.)

If an error occurs, it will not be detected. The result of the error-producing operation is undefined.

Safety 1

```
(cmi::error-if-location cm:overflow-flag 66575)
```

This safety level checks for errors, but it does not report an error until the next time a value is read from the CM by *Lisp functions such as **pref**, ***max**, and ***sum**. Because the error is not reported immediately, normal debugging tools cannot be used to find it. The only information that would be reported in the example above is that an integer **+!!** expression produced a value that was too large.

Safety 2

```
(slc::error-if-location cm:overflow-flag 66575 u8)
```

The behavior of this safety level depends on the value of the variable ***immediate error if location***. This variable is set by the option **Immediate Error If Location**. If it variable is **t**, **Safety 2** behaves like **Safety 3**; if this variable is **nil**, **Safety 2** behaves like **Safety 1**. **Safety 2** allows greater flexibility for debugging by making a run-time check. For example, code expected to have few bugs can be run with this variable set to **nil**, that is, at safety level 1. Then, if it produces an error, the code can be run again with the variable set to **t**, that is, at safety level 3. Reproducing the error using this method does not require recompilation.

Safety 3

```
(if (plusp (cm:global-logior cm:overflow-flag 1))
    (slc::pvar-error cm:overflow-flag 66575 u8))
```

This safety level checks for errors and reports any found error immediately. A report includes enough information to describe the processors that have the error and to display the values in those processors. For example, suppose the code example given at the beginning of this section is modified to include the first two ***set** assignments shown below.

```
(proclaim '(type (field-pvar 8) u8 u8-2)
(*set u8 (load-byte!! (self-address!!) (!! 0) (!! 8)))
(*set u8-2 (!! 3))
(*set u8 (+!! u8 u8-2))
```

The error detection code produces an error message similar to the one below. (This example message is shown as it appears to users on a UNIX front end in the Lucid Common Lisp environment. The message is the same on the Symbolics Lisp machine front end except that different key sequences are given to invoke the proceed options.)

```
>>Error: In +!!.
The result of a (two argument) integer +!! is too big for
its destination. There are 512 selected processors, 6 proc-
essors have an error. The object that caused the overflow
is a (UNSIGNED-BYTE 8) pvar named U8.

:A    Abort to Lisp Top Level
:C    Ignore error.
:@ 0  Ignore Error.
:@ 1  Display Processors With Error.
:@ 2  Display Value in Processors with Error.
:@ 3  Display Selected Processors.
:@ 4  Display Value in Selected Processors.
:@ 5  Display Value in All Processors.

-> :@ 1
:@ 1  Display Processors With Error.

253 254 255 509 510 511

-> :@ 2
:@ 2  Display Value in Processors with Error.

253 = 0
254 = 1
255 = 2
509 = 0
510 = 1
511 = 2

-> :c
Ignore error.
```

```
NIL
>
```

3.3.1 Functions Reporting Errors

For the *Lisp operations ***set**, ***setf**, and ***pset**, if the destination field is smaller than the result of the source expression, an error is reported. The following functions are representative of *Lisp functions that report errors unrelated to the size of the destination field.

lognot!!
Within a ***set**, a **lognot!!** expression can get an error when a negative result is produced for an unsigned destination.

+!! ***!!**
-!! **/!!**
An error occurs for floating point overflow. For **/!!** an error also occurs for division by zero.

ceiling!! **mod!!**
truncate!! **rem!!**
round!! **floor!!**
An error occurs for division by zero.

sqrt!!
isqrt!!
An error occurs for negative numbers. (Note that **sqrt!!** does not produce a complex result when given a negative pvar; **sqrt!!** only returns a complex pvar if given a complex input pvar.)

float!!

An error occurs when coercing an integer larger than the specified float format, or when coercing a float to a smaller format.

pref!!
***pset**

An error occurs when **pref!!** attempts to get a value from a processor or when ***pset** attempts to write to a processor that is not on the grid.

4 Setting Compiler Options

The compiler options control the behavior of the *Lisp compiler, including the degree of optimization it performs while generating code. There are two ways to set the compiler options: using a menu and setting the *Lisp variables directly.

4.1 Using the Compiler Options Menu

The simplest method of setting *Lisp compiler options is to invoke the *Lisp compiler options menu as detailed below.

The options menu must be invoked from within the *Lisp package. First, ensure that the *Lisp package is the current package by the executing the form:

```
(in-package '*lisp)
```

Next, type the following to a Lisp Listener or at Lisp top level:

```
(compiler-options)
```

The default *Lisp compiler options menu is displayed.

Two alternate methods of invoking the *Lisp compiler options menu are available on a Symbolics Lisp machine front end. With the *Lisp package loaded:

- To a Lisp Listener, type:

  ```
  :Set Compiler Options
  ```

- In the editor, press **Meta-x** and type:

  Set Compiler Options

The *Lisp compiler options menu lists the following options. Default values are shown in boldface type.

```
Starlisp Compiler Options

Compile Expressions (Yes, or No)
Warning Level (High, Normal, None)
Inconsistency Reporting Action (Abort, Error, Cerror, Warn, None)
Safety (0, 1, 2, 3)
Print Length for Messages (an integer, or Nil) 4
Print Level for Messages (an integer, or Nil) 3
Pull Out Common Address Expressions (Yes, or No)
Use Always Instructions (Yes, or No)
```

On the Symbolics front end, changes are made by clicking the mouse cursor and by entering values where appropriate. To change an option value, move the mouse cursor over the desired value and click the left mouse button. The value selected is displayed in bold. To exit the menu and save the selections made, click the left mouse button on the small box marked **Exit**. To exit the menu without saving the new selections, click on the small box marked **Abort**.

On a UNIX front end, options are listed one at a time—each with its current value. The system waits for user input before listing the next option. To keep the current value and go on to the next option, press the Return key. To change the value, type the desired value and press the Return key. At the end of the options list, confirmation is requested:

```
Do the assignment? (Yes, or No)
>
```

To save any changes made, simply press the key or type **Yes** and press the Return key. To cancel the changes made, type **No** and press the Return key.

Not all available options for controlling the behavior of the *Lisp compiler are listed by default when the options menu is invoked. The options that are not in the default menu provide capabilities that are not generally needed.

To invoke the options menu with all options listed, type the following at Lisp top level or to a Lisp Listener:

```
(compiler-options :class :all)
```

Alternately, to a Lisp Listener on a Symbolics Lisp machine front end, type:

:Set Compiler Options :class all

The full options menu lists the following compiler options. Default values are shown in boldface type.

Starlisp Compiler Options

Compile Expressions (**Yes**, or No)
Warning Level (High, **Normal**, None)
Inconsistency Reporting Action (Abort, Error, Cerror, **Warn**, None)
Safety (0, **1**, 2, 3)
Print Length for Messages (**an integer**, or Nil) **4**
Print Level for Messages (**an integer**, or Nil) **3**
Optimize Bindings (No, **Cspeed<3**, Yes)
Peephole Optimize Paris (No, **Cspeed<3**, Yes)
Pull Out Common Address Expressions (Yes, or **No**)
Use Always Instructions (Yes, or **No**)
Machine Type (**Current**, Compatible, Cm1, Cm2, Cm2-FPA, Simulator)
Add Declares (Everywhere, Yes, **No**)
Use Undocumented Paris (**Yes**, or No)
Verify Type Declarations (No, **Current-Safety**, Yes)
Constant Fold Pvar Expressions (**Yes**, or No)
Speed (0, **1**, 2, 3)
Compilation Speed (0, **1**, 2, 3)
Space (0, **1**, 2, 3)
Strict THE Type (**Yes**, or No)
Immediate Error If Location (**Yes**, or No)
Optimize Check Stack Expression (**Yes**, or No)
Generate Comments With Paris Code (**Yes**, Macro, No)

4.2 Setting *Lisp Compiler Variables Directly

Instead of using the *Lisp compiler options menu as described in the preceding section, *Lisp compiler options may be changed by changing the associated *Lisp variables, or, for certain options, by using a global declaration. The forms to use to set compiler option variables directly are: **compiler-let**, **optimize**, ***optimize**, and **declare**. These are described below. Section 5 describes each compiler option and gives the name of the *Lisp variable associated with it.

compiler-let

The Common Lisp special form **compiler-let** can be used to selectively change the value of any *Lisp compiler option for a region of code. This is demonstrated by the following code fragment.

```
(compiler-let ((*compilep* t) (*safety* 0)
               (*use-always-instructions* t))
  (code to compile at low safety))
```

The above form ensures that the *Lisp compiler is enabled and sets the *Lisp compiler safety level to 0 and enables the use of Paris **always** instuctions for the region of code enclosed by the **compiler-let** form.

optimize

The Common Lisp **optimize** declaration specifier may be used within either a ***proclaim** form or a **declare** form to change optimization levels for *both* the Common Lisp compiler and *Lisp compiler. The following qualities may be set in this way:

safety	**speed**
space	**compilation-speed**

For example,

```
(*proclaim '(optimize (safety value)))
```

globally sets the safety level to *value* for both compilers.

*optimize

The ***optimize** declaration specifier, used within a ***proclaim** or a **declare** form, changes the optimization level for the *Lisp compiler only; it does not affect the Common Lisp compiler. The following qualities may be set in this way:

safety	**speed**
space	**compilation-speed**

For example,

```
(*proclaim '(*optimize (safety value)))
```

globally sets the *Lisp safety level to *value.*

declare

The **declare** form may be used with either the **optimize** or the ***optimize** declaration specifier to change the *Lisp optimization levels. This should be done from within either a ***defun**, a ***let**, a ***let***, or a ***locally** form. For example,

```
(*let ((truth t!!))
  (declare (optimize (safety 3)))
  (foo (bar truth)))
```

sets both the Common Lisp and the *Lisp safety levels at 3 for the entire body of the ***let** form.

5 Compiler Options Reference

A list of all the available *Lisp compiler options is given below. The default value for each option is given in boldface type. A description of each option follows the list.

Starlisp Compiler Options

Compile Expressions (**Yes**, or No)
Warning Level (High, **Normal**, None)
Inconsistency Reporting Action (Abort, Error, Cerror, **Warn**, None)
Safety (0, **1**, 2, 3)
Print Length for Messages (**an integer**, or Nil) **4**
Print Level for Messages (**an integer**, or Nil) **3**
Optimize Bindings (No, **Cspeed<3**, Yes)
Peephole Optimize Paris (No, **Cspeed<3**, Yes)
Pull Out Common Address Expressions (Yes, or **No**)
Use Always Instructions (Yes, or **No**)
Machine Type (**Current**, Compatible, Cm1, Cm2, Cm2-FPA, Simulator)
Add Declares (Everywhere, Yes, **No**)
Use Undocumented Paris (**Yes**, or No)
Verify Type Declarations (No, **Current-Safety**, Yes)
Constant Fold Pvar Expressions (**Yes**, or No)
Speed (0, **1**, 2, 3)
Compilation Speed (0, **1**, 2, 3)
Space (0, **1**, 2, 3)
Immediate Error If Location (**Yes**, or No)
Optimize Check Stack Expression (Yes, or **No**)
Generate Comments With Paris Code (**Yes**, Macro, No)

Compile Expressions

Values: **Yes (t), No (nil)**
Default: **Yes (t)**
Variable: ***compilep***

The **Compile Expressions** option enables or disables the *Lisp compiler. A value of **Yes (t)** enables the compiler; a value of **No (nil)** disables it.

The *Lisp compiler is on by default.

Warning Level

Values: **High (:high), Normal (:normal), None (:none)**
Default: **Normal (:normal)**
Variable: ***warning-level***

The **Warning Level** option controls the kind of warnings produced by the *Lisp compiler.

A warning level value of **High (:high)** causes the compiler to generate a warning whenever an expression is not compiled. The warning tries to explain why the expression is not compiled. Usually the cause is lack of type information, as shown in the following example:

```
(*proclaim '(type (pvar (signed-byte 8)) s8))
(*set s8 (+!! s8 variable))
```

Attempting to compile the above code with the warning level set to **High (:high)**, produces the following warning:

```
;;; Warning: Verbose: While compiling VARIABLE:
The expression (*LISP-I::*SET-1 S8 (+!! S8 VARIABLE)) is
not compiled because +!! does not understand undeclared ex-
pressions.
```

In contrast, the following can be successfully compiled because the type of **variable** is supplied.

```
(*proclaim '(type (pvar (signed-byte 8)) s8))
(*set s8 (+!! s8 (the (pvar (signed-byte 8)) variable)))
```

A warning level value of **Normal (:normal)**, the default, causes the compiler to generate warnings only for invalid arguments and type mismatches.

If the first code example above is compiled with the warning level set to **Normal (:normal)**, the ***set** expression is not compiled, but no warning is given. On the other hand, with warning level set to **Normal (:normal)**, an attempt to compile

```
(*proclaim '(type (pvar (field-pvar 8) u8))
(*proclaim '(type boolean-pvar b1))
(*set u8 (-!! b1))
```

results in this warning:

```
Warning: While compiling B1:
Function -!! expected a numeric pvar argument but got a
boolean pvar argument.
```

A warning level value of **None (:none)** prevents the compiler from generating any warnings.

Inconsistency Reporting Action

Values:	**Abort (:abort), Error (:error), Cerror(:cerror), Warn (:warn), None (:none)**
Default:	**Warn (:warn)**
Variable:	***inconsistency-action***

The **Inconsistency Reporting Action** option controls the behavior of the compiler when an inconsistency is discovered. An inconsistency usually indicates an implementation error in the compiler.

An value of **Abort (:abort)** causes the compiler to report a discovered compiler inconsistency and immediately abort the compilation.

A value of **Error (:error)** causes the compiler to report a discovered compiler inconsistency using the Common Lisp function **error**. This signals a fatal error from which it is impossible to continue and enters the debugger.

A value of **Cerror (:cerror)** causes the compiler to report a discovered compiler inconsistency using the Common Lisp function **cerror**. This signals a continuable error and enters the debugger. The program may be resumed after the error is resolved.

A value of **Warn (:warn)** causes the compiler to report a discovered compiler inconsistency using the Common Lisp function **warn**. This prints an error message but normally does not enter the debugger.

A value of **None (:none)** prevents the compiler from taking any action when an inconsistency in the compiler is discovered.

Safety

Values:	**0, 1, 2, 3,**
Default:	**1**
Variable:	***safety***

The **Safety** option controls what kind of code the compiler generates to detect error conditions and how error conditions are reported.

A value of **0**, termed *low safety*, prevents error conditions from being signaled.

A value of **1**, causes error conditions to be signaled, but error notification does not occur at the time of the error.

A value of **2 or 3**, termed *high safety*, causes error conditions to be signaled by error messages that attempt to be as helpful as possible.

In general, low safety produces the fastest and most dangerous code. For a more detailed discussion recommending when to use each of the four safety levels, see Section 3.3, entitled "Safety Optimization and Error Detection."

Print Length for Messages
Print Level for Messages

Values:	**<an integer>, or nil**	
Length Default:	**4**	
Level Default:	**3**	
Variables:	***slc-print-length***	***slc-print-level***

These options control how much of an expression the compiler prints when generating a warning about that expression. As in Common Lisp, the **Print Level** indicates how many levels of data object nesting will be printed, counting from 0. The **Print Length** indicates how many elements at each level will be printed, counting from 1. For both variables, if the value **nil** is specified, all elements at all levels are printed. The Common Lisp variables ***print-length*** and ***print-level*** are bound to these variables when messages are printed.

Optimize Bindings

Values:	**No (nil), Cspeed<3 (cspeed<3), Yes (t)**
Default:	**Cspeed<3 (cspeed<3)**
Variable:	***optimize-bindings***

The **Optimize Bindings** option provides control over compilation speed by altering the number of temporary bindings generated by the *Lisp compiler.

A value of **Yes (t)** enables this option and causes extra bindings to be removed. When binding optimization is enabled, some temporary variables are eliminated and others are used repeatedly.

A value of **No (nil)** disables binding optimization. When the binding optimization option is disabled, the code produced by the compiler is more readable because it uses unique temporary address variables to represent each value represented.

A value of **Cspeed<3** varies binding optimization based on the value of the compilation speed variable ***compilation-speed***. If compilation speed is 3 (the highest possible value), then ***optimize-bindings*** is set to **nil**. If compilation speed is less than 3, then ***optimize-bindings*** is set to **t**.

Peephole Optimize Paris

Values:	**No (nil), Cspeed<3 (3), Yes (t)**
Default:	**Cspeed<3 (3)**
Variable:	***optimize-peephole***

The **Peephole Optimize Paris** option controls the *Lisp compiler's peephole optimization of generated Lisp/Paris code.

A value of **Yes (t)** causes the *Lisp compiler to optimize the Lisp/Paris code it generates. A value of **No (nil)** prevents this optimization.

A value of **Cspeed<3** varies peephole optimization based on the value of the compilation speed variable ***compilation-speed***. If compilation speed is 3 (the highest possible value), then ***optimize-peephole*** is set to **nil**. If compilation speed is less than 3, then ***optimize-peephole*** is set to **t**.

Pull Out Common Address Expressions

Values: **Yes (t), No (nil)**
Default: **No (t)**
Variable: ***pull-out-subexpressions***

The **Pull Out Common Address Expressions** option determines whether the compiler performs common subexpression elimination on address expressions such as calls to **pvar-location**. Enabling this option can, in certain circumstances, increase performance significantly.

A value of **Yes (t)** enables this optimization; a value of **No (nil)** disables it.

When enabled, this option trims the code executed on the front end; it does not affect the code executed on the Connection Machine. If a program already has a high Connection Machine utilization, this option will do little to improve the execution time. Conversely, if a program has a low Connection Machine utilization, enabling **Pull Out Common Address Expressions** can reduce execution time. The potential benefit is usually greater for larger expressions, where there are more opportunities for common addressing expressions.

For example, consider the following ***set** expression:

```
(*set s16 (+!! (*!! s8 s8-2) s16-2))
```

Here is the code produced by the compiler with this option *enabled*:

```
(let* ((pvar-location-s16-1 (pvar-location s16))
       (pvar-location-s8-2 (pvar-location s8))
       (pvar-location-s8-2-3 (pvar-location s8-2))
       (pvar-location-s16-2-4 (pvar-location s16-2)))
  (cm:multiply pvar-location-s16-1 pvar-location-s8-2
               pvar-location-s8-2-3 16 8 8)
  (cm:+ pvar-location-s16-1 pvar-location-s16-2-4 16)
  (cmi::error-if-location cm:overflow-flag 66575)
  nil)
```

Here is the code produced with this option *disabled*:

```
(progn
  (cm:multiply (pvar-location s16)
               (pvar-location s8)
               (pvar-location s8-2) 16 8 8)
  (cm:+ (pvar-location s16) (pvar-location s16-2) 16)
  (cmi::error-if-location cm:overflow-flag 66575)
  nil)
```

Notice that **pvar-location** is executed four times when **Pull Out Common Address Expressions** is enabled and five times when it is disabled.

Use Always Instructions

Values:	**Yes (t), No (nil)**
Default:	**No (nil)**
Variable:	***use-always-instructions***

The **Use Always Instructions** option determines whether or not the *Lisp compiler generates unconditional **Always** Paris instructions for stack operations.

A value of **Yes (t)** enables the use of the Paris **Always** instructions; a value of **No (nil)** disables their use.

This option is not fully implemented and may generate undocumented Paris instructions. It is more useful to users of a CM-2 with the floating-point accelerator than to other users.

For an example of code generated when this option is set to **Yes**, see the last example under the **Machine Type** option description.

Machine Type

Values:	**Current (:current), Compatible (:compatible), CM1 (:cm1), CM2 (:cm2), CM2-FPA (:cm2-fpa), Simulator (:simulator)**
Default:	**Current (:current)**
Variable:	***machine-type***

The **Machine Type** option directs the *Lisp compiler to generate code that is either specific to one of the Connection Machine models or compatible across models.

A value of **Current (:current)** allows the compiler to generate code specific to the current machine type.

A value of **Compatible (:compatible)** allows the compiler to generate code compatible across machine types.

A value of **CM1 (:cm1)** allows the compiler to generate code specific to Connection Machine model CM-1.

A value of **CM2 (:cm2)** allows the compiler to generate code specific to the CM-2.

A value of **CM2-FPA (:cm2-fpa)** allows generation of code specific to the CM-2 with the floating-point accelerator. When machine type CM2-FPA is specified, the *Lisp compiler generates Paris instructions that take advantage of the floating point accelerator hardware. This is the most useful value of the machine type option.

A value of **Simulators** allow the compiler to generate code specific to the simulator. This value is currently equivalent to the **Compatible** setting.

The example below demonstrates how the **Machine Type** option interacts with other compiler options. Code generated by compiling a ***sum** expression using three different combinations of the **Machine Type** and **Use Always Instructions** option settings is shown. Each successive combination produces more efficient code than the last. In each, the **safety** option is set to `0` to eliminate error detection code and to make the examples more readable.

Consider the following *Lisp code:

```
(*proclaim '(type (pvar single-float) sf1 sf2))
(*sum (*!! (+!! sf1 (!! 128.0)) sf2))
```

When the **Machine Type** option is set to **Compatible (:compatible)** and the **Use Always Instructions** option is set to **No (nil)**, the compiler generates the following code:

```
(let* ((slc::old-next-stack-field (cm:allocate-stack-field 32))
       (*!!-index-2 (+ slc::old-next-stack-field 32)))
  (declare (ignore *!!-index-2))
  (prog1
    (progn
      ;; Move constant - !!.

      (cm:move-constant slc::old-next-stack-field 1124073472 32)
      (cmi::clear-mem cm:overflow-flag)
      (cm:f-add-2-11 slc::old-next-stack-field
        (pvar-location sf1) 23 8)
      ;; The result of a (two argument) float +!! overflowed.
      (cmi::error-if-location cm:overflow-flag 394259 nil)
      (cm:f-multiply-2-11 slc::old-next-stack-field
        (pvar-location sf2) 23 8)
      ;; The result of a (two argument) float *!! overflowed.
      (cmi::error-if-location cm:overflow-flag 394003)
      (cmi::global-float-add slc::old-next-stack-field 23 8))
      (cm:deallocate-upto-stack-field slc::old-next-stack-field)))
```

However, when **Machine Type** is set to **CM2-FPA (:cm2-fpa)** and **Use Always Instructions** is set to **No (nil)**, the compiler generates the following, more efficient, code:

```
(let* ((slc::old-next-stack-field (cm:allocate-stack-field 32))
       (*!!-index-2 (+ slc::old-next-stack-field 32)))
  (declare (ignore *!!-index-2))
  (prog1
    (progn
      (cmi::clear-mem cm:overflow-flag)
      (cm:f-add-constant-3-11 slc::old-next-stack-field
        (pvar-location sf1) 128.0 23 8)
      ;; The result of a (two argument) float +!! overflowed.
      (cmi::error-if-location cm:overflow-flag 394259 nil)
      (cm:f-multiply-2-11 slc::old-next-stack-field
        (pvar-location sf2) 23 8)
      ;; The result of a (two argument) float *!! overflowed.
      (cmi::error-if-location cm:overflow-flag 394003)
      (cmi::global-float-add slc::old-next-stack-field 23 8))
      (cm:deallocate-upto-stack-field slc::old-next-stack-field)))
```

The most efficient code is generated when **Machine Type** is set to **CM2-FPA (:cm2-fpa)** and **Use Always Instructions** is set to **Yes (t)**:

```
(let* ((slc::old-next-stack-field (cm:allocate-stack-field 32))
       (*!!-index-2 (+ slc::old-next-stack-field 32)))
  (declare (ignore *!!-index-2))
  (prog1
    (progn
      (cmi::clear-mem cm:overflow-flag)
      (cm:f-add-const-always-3-11 slc::old-next-stack-field
        (pvar-location sf1) 128.0 23 8)
      ;; The result of a (two argument) float +!! overflowed.
      (cmi::error-if-location cm:overflow-flag 394259 nil)
      (cm:f-multiply-always-2-11 slc::old-next-stack-field
        (pvar-location sf2) 23 8)
      ;; The result of a (two argument) float *!! overflowed.
      (cmi::error-if-location cm:overflow-flag 394003)
(cmi::global-float-add slc::old-next-stack-field 23 8))
(cm:deallocate-upto-stack-field slc::old-next-stack-field)))
```

Add Declares

Values: **Everywhere (:everywhere), Yes (t), No (nil)**
Default: **No (nil)** on symbolics, **Yes (t)** on other front ends
Variable: ***add-declares***

The **Add Declares** compiler option determines if and how the *Lisp compiler will generate code that includes type declarations for stack address computations.

A value of **Everywhere (:everywhere)** causes the compiler to generate type declarations using both **declare** and **the** forms. A **the** form is used wherever a **declare** form is not legal.

A value of **Yes (t)** causes the compiler to generate type declarations wherever a **declare** form is appropriate.

A value of **No (nil)** prevents the compiler from generating any type declarations. The **Add Declares** option default value on Symbolics Lisp machines is **nil** because Symbolics' implementation generally ignores type declarations.

Consider the following code.

```
(*sum (if!! b1 s8 s8-2))
```

With the **Add Declares** option set to **Yes (t)**, this generates the following Lisp/Paris code.

```
(let* ((slc::old-next-stack-field (cmi::next-stack-field))
       (if!!-context-1 (+ slc::old-next-stack-field 8))
       (if!!-index-2 (+ if!!-context-1 1)))
  (declare
    (type slc::cm-address slc::old-next-stack-field
      if!!-context-1 if!!-index-2))
  (prog1
    (progn
      (cm:allocate-stack-field
        (- if!!-index-2 slc::old-next-stack-field))
      (cmi::store-context-always if!!-context-1)
      (cmi::load-context (pvar-location b1))
      (cm:move slc::old-next-stack-field (pvar-location s8) 8)
      (cm:logandc1-always cm:context-flag if!!-context-1 1)
    (cm:move slc::old-next-stack-field (pvar-location s8-2) 8)
    (cmi::load-context-always if!!-context-1)
    (cm:global-add slc::old-next-stack-field 8))
  (cm:deallocate-upto-stack-field slc::old-next-stack-field)))
```

Notice that there is one **declare** form in the compiler output above.

Use Undocumented Paris

Values: **Yes (t), No (nil)**
Default: **Yes (t)**
Variable: ***use-undocumented-paris***

The **Use Undocumented Paris** compiler option determines whether or not the code generated by the *Lisp compiler uses undocumented Paris instructions.

A value of **Yes (t)** allows the use of undocumented Paris instructions. In many cases, enabling this option significantly increases the execution speed of compiled *Lisp code.

A value of **No (nil)** disallows the use of most undocumented Paris instructions.

For example, with **Use Undocumented Paris** set to Yes (t), compiling

```
(*sum (if!! b1 s8 s8-2))
```

results in code that includes three internal, undocumented Paris functions in the CMI package. When the same ***sum** statement is compiled with this option set to **No (nil)**, the following code is generated. It includes only documented functions in the CM package.

```
(let* ((slc::old-next-stack-field (cmi::next-stack-field))
       (if!!-context-1 (+ slc::old-next-stack-field 8))
       (if!!-index-2 (+ if!!-context-1 1)))
  (prog1
    (progn
      (cm:allocate-stack-field
        (- if!!-index-2 slc::old-next-stack-field))
      (cm:move-always if!!-context-1 cm:context-flag 1)
      (cm:move cm:context-flag (pvar-location b1) 1)
      (cm:move slc::old-next-stack-field (pvar-location s8) 8)
      (cm:logandc1-always cm:context-flag if!!-context-1 1)
      (cm:move slc::old-next-stack-field (pvar-location s8-2) 8)
      (cm:move-always cm:context-flag if!!-context-1 1)
    (cm:global-add slc::old-next-stack-field 8))
  (cm:deallocate-upto-stack-field slc::old-next-stack-field)))
```

If the **Use Undocumented Paris** option is disabled, it still allows the *Lisp compiler to generate undocumented Paris routines in cases where no appropriate documented Paris instructions exists. However, if a documented instruction exists, it will be used—even if the undocumented instruction is faster.

Verify Type Declarations

Values:	**No (nil), Current-Safety (:current-safety), Yes (t) or an integer between 0 and 3**
Default:	**Current-Safety (:current-safety)**
Variable:	***verify-type-declarations***

The **Verify Type Declaration** compiler option determines whether or not the *Lisp compiler generates type verification code for arguments to ***defun** functions that have been given either **the** or **declare** type declarations.

This option is primarily used while debugging *Lisp programs. The most common user errors are declaring pvar arguments incorrectly and violating type declarations. These errors are often hard to track down because the results of violating a type declaration can be unpredictable. With the safety option set at 3, and the **Verify Type Declarations** option enabled, the compiler generates code to catch erroneous and violated type declarations immediately.

An **integer** value sets verification to a nonexistent, low, high, or intermediate level. At a setting of **0**, no error checking is done. At a setting of **1**, a minimal amount of error checking is done. At a setting of **2**, a moderate amount of error checking is done. The highest setting is **3**, which causes the *Lisp compiler to generate code for maximal type verification error checking.

A value of **Yes (t)** causes to the compiler to generate a maximum amount of error checking and is equivalent to a value of 3.

A value of **No (nil)** prevents the compiler from generating any type verification code and is equivalent to a value of 0.

A value of **Current-Safety (:current-safety)** sets the verification level based on the safety level. If the **Safety** option is set to 0, and **Verify Type Declarations** is set to **Current-Safety**, no verification code is generated. With safety at 3, verification becomes likewise set to 3, and so on.

As an example, consider the following `*sum` expression.

```
(*sum (the (field-pvar 32) quux)))
```

A verification value of **0** causes the compiler to generate the least amount of type checking code: none. At verification level 0, this ***sum** expression compiles into the following:

```
(cm:global-unsigned-add (pvar-location quux) 32)
```

A value of **1** causes the compiler to generate a little bit of error checking:

```
(progn (if (not (*lisp-i:internal-pvarp quux))
             (slc::error-doesnt-match-declaration
               quux '(pvar (unsigned-byte 32))))
       (cm:global-unsigned-add (pvar-location quux) 32))
```

Above, a check is done to make sure that the variable **quux** is a pvar.

A value of **2** causes the compiler to generate more error checking:

```
(progn (if (not (and (*lisp-i:internal-pvarp quux)
                     (eq (pvar-type quux) :field)))
             (slc::error-doesnt-match-declaration
                quux '(pvar (unsigned-byte 32))))
       (cm:global-unsigned-add (pvar-location quux) 32))
```

Here, verification code ensures that the variable **quux** is a field pvar.

A value of **3** causes the compiler to generate maximum error checking code:

```
(progn (if (not (and (*lisp-i:internal-pvarp quux)
                     (eq (pvar-type quux) :field)
                     (eql (pvar-length quux) 32)))
             (slc::error-doesnt-match-declaration
                quux '(pvar (unsigned-byte 32))))
       (cm:global-unsigned-add (pvar-location quux) 32))
```

The code above checks to make sure that the variable **quux** is a field pvar of length 32.

Constant Fold Pvar Expressions

Values: **Yes (t), No (nil)**
Default: **Yes (t)**
Variable: ***constant-fold***

The **Constant Fold Pvar Expressions** compiler option determines whether or not the *Lisp compiler will constant fold certain pvar expressions.

A value of **Yes(t)** allows the compiler to constant fold pvar expressions in which all arguments to certain *Lisp functions contain identical values in all active processors. Examples of the arguments threated are: **nil!!**, **t!!**, and calls to the function **!!**.

A value of **No (nil)** prevents the compiler from constant folding.

For example, with constant folding enabled,

```
(*sum (-!! (!! 1.0))))
```

compiles into:

```
(progn
  ;; Constant global sum - *sum.
  (* -1.0 (cm:global-count-always cm:context-flag)))
```

In contrast, without constant folding, the same ***sum** expression compiles into:

```
(let* ((slc::old-next-stack-field (cmi::next-stack-field))
       (-!!-index-2 (+ slc::old-next-stack-field 32)))
  (prog1
    (progn
      (cm:allocate-stack-field
        (- -!!-index-2 slc::old-next-stack-field))
      ;; Move constant - !!.

      (cm:move-constant slc::old-next-stack-field 1065353216 32)
      (cm:lognot (+ slc::old-next-stack-field 31)
                 (+ slc::old-next-stack-field 31) 1)
      (cmi::global-float-add slc::old-next-stack-field 23 8))
    (cm:deallocate-upto-stack-field slc::old-next-stack-field)))
```

Clearly, constant folding allows the compiler to generate more efficient code.

Speed

Values:	**0, 1, 2, 3**
Default:	**1**
Variable:	***speed***

The **Speed** compiler option advises both the Common Lisp and the *Lisp compilers of the relative importance of the quality **speed.**

A value of **0**, known as *low speed,* means speed of execution is totally unimportant.

A value of **1**, the default, means speed of execution is of little importance.

A value of **2** means speed of execution is of moderate importance.

A value of **3** means speed of execution is extremely important.

Compilation Speed

Values:	**0, 1, 2, 3**
Default:	**1**
Variable:	***compilation-speed***

The **Compilation Speed** compiler option advises both the Common Lisp and the *Lisp compilers of the relative importance of the quality **compilation-speed.**

A value of **0**, known as *low compilation speed,* means speed of compilation is totally unimportant.

A value of **1**, the default, means speed of compilation is of little importance.

A value of **2** means speed of compilation is of moderate importance.

A value of **3** means speed of compilation is extremely important. If this value is set, both the **Optimize Peephole** and **Optimize Bindings** options are disabled.

Space

Values:	**0, 1, 2, 3**
Default:	**1**
Variable:	***space***

The **Space** compiler option advises both the Common Lisp and the *Lisp compilers of the relative importance of the quality **space**. The **space** quality governs both code size and run-time space utilization.

A value of **0**, means code bulk and instruction space utilization are totally unimportant.

A value of **1**, the default, means code bulk and instruction space utilization are of little importance.

A value of **2** means code bulk and instruction space utilization are of moderate importance.

A value of **3** means code bulk and instruction space utilization are extremely important.

Immediate Error If Location

Values:	**Yes (t), No (nil)**
Default:	**Yes (t)**
Variable:	***immediate-error-if-location***

The **Immediate Error If Location** compiler option determines how a **Safety** option setting of **2** behaves.

A value of **Yes (t)** makes **Safety 2** behave like **Safety 3**.

A value of **No (nil)** makes **Safety 2** behave like **Safety 1**.

See section 3.3 for a discussion of safety levels.

Optimize Check Stack Expression

Values:	**Yes (t), No (nil)**
Default:	**Yes (yes)**
Variable:	***optimize-check-stack***

The **Optimize Check Stack Expression** compiler option determines how the *Lisp compiler manages the temporary stack space used by the Lisp/Paris code it generates.

A value of **Yes (t)** makes the compiler try to remove the length expression in a call to **cm:allocate-stack-field** generated when the *compiled code uses stack space.

A value of **No (nil)** disables this optimization.

Generate Comments With Paris Code

Values:	**Yes (t), Macro (:macro), No (nil)**
Default:	**Yes (t)**
Variable:	***generate-comments***

The **Generate Comments With Paris Code** compiler option controls whether or not the *Lisp compiler inserts comments into the Lisp/Paris code it generates.

A value of **Yes (t)** causes the compiler to generate comments

A value of **Macro (:macro)** causes the compiler to generate comments when forms are being macroexpanded using the Symbolics editor command **Macro Expand Expression.**

A value of **No (nil)** prevents the compiler from generating annotated Lisp/Paris code.

With the **Generate Comments With Paris Code** option set to **Yes (t)**, compiling

```
(*sum (if!! b1 s8 s8-2))
```

generates the following:

```
(let* ((slc::old-next-stack-field (cmi::next-stack-field))
       (if!!-context-1 (+ slc::old-next-stack-field 8))
```

```
      (if!!-index-2 (+ if!!-context-1 1)))
  (prog1
    (progn
      (cm:allocate-stack-field
        (- if!!-index-2 slc::old-next-stack-field))
      ;; Save context - if!!.

      (cmi::store-context-always if!!-context-1)
      ;; Compute predicate, and select processors
      ;; where the predicate is true. - if!!.
      (cmi::load-context (pvar-location b1))
      ;; Compute and move then clause in context of processors
      ;; where the predicate is true. - if!!.
      (cm:move slc::old-next-stack-field (pvar-location s8) 8)
      ;; Select processors where the predicate is false. - if!!.

      (cm:logandc1-always cm:context-flag if!!-context-1 1)
      ;; Compute and move else clause in context of processors
      ;; where the predicate is false. - if!!.
      (cm:move slc::old-next-stack-field (pvar-location s8-2) 8)
      ;; Restore context - if!!.

      (cmi::load-context-always if!!-context-1)
      (cm:global-add slc::old-next-stack-field 8))
    (cm:deallocate-upto-stack-field slc::old-next-stack-field)))
```

With the **Generate Comments With Paris Code** option set to **No (nil)**, compiling

(*sum (if!! b1 s8 s8-2))

generates the following:

```
(let* ((slc::old-next-stack-field (cmi::next-stack-field))
       (if!!-context-1 (+ slc::old-next-stack-field 8))
       (if!!-index-2 (+ if!!-context-1 1)))
  (prog1
    (progn
      (cm:allocate-stack-field
        (- if!!-index-2 slc::old-next-stack-field))
      (cmi::store-context-always if!!-context-1)
      (cmi::load-context (pvar-location b1))
      (cm:move slc::old-next-stack-field (pvar-location s8) 8)
      (cm:logandc1-always cm:context-flag if!!-context-1 1)
    (cm:move slc::old-next-stack-field (pvar-location s8-2) 8)
    (cmi::load-context-always if!!-context-1)
    (cm:global-add slc::old-next-stack-field 8))
  (cm:deallocate-upto-stack-field slc::old-next-stack-field)))
```

Part II
*Lisp Compiler Practicum

6 Tricks of the Trade

This part of the *Lisp Compiler Guide is a narrative tutorial that explains how to use the *Lisp compiler. This tutorial is not a substitute for the first part of the *Guide to the *Lisp Compiler*. Part I, "*Lisp Compiler Features," should be read first and referred to as necessary during this tutorial. Here we address the following issues:

1. How the *Lisp compiler works
2. How to use **declare**, ***proclaim**, **the**, and ***locally** forms.
3. How to write general *Lisp code that compiles.

This tutorial will *not* teach you how to write efficient *Lisp code. Also, it does not provide instruction in analyzing the Lisp/Paris output produced by the *Lisp compiler to determine whether it makes efficient use of the Connection Machine system.

With a little practice you will be able to solve the problems presented by these two commonly asked questions:

Question: How can I write a *Lisp program so that the *Lisp compiler will not report that it cannot compile certain statements, but will instead compile the entire program?

Question: Given a *Lisp program written by someone who couldn't answer the first question, how can I quickly turn it into an equivalent *Lisp program that will compile?

To accomplish this task, the reader is strongly advised to try all the examples given here.

The compilation process effected by the *Lisp compiler is unlike that of most compilers. The *Lisp compiler executes as part of the Common Lisp compiler, and it does not attend to every statement in a *Lisp program. To avoid ambiguity, therefore, the terms **compiler*, **compiling*, **compilation*, and **compiled code* are used when referring to the *Lisp compiler, the process by which it translates *Lisp code into Lisp/Paris, and the results of that process, respectively. This is done to make a clear distinction between normal Common Lisp compilation and *Lisp compilation.

7 What Makes the *Lisp Compiler Work?

To demonstrate what makes the *Lisp compiler work, we walk through the process of taking a small, uncompilable piece of *Lisp code and changing it so the *Lisp compiler can handle it.

Unlike other compilers, which translate entire programs into internal representations or lower level languages, the *Lisp compiler does not translate any and all legal *Lisp code into Lisp/Paris. The *Lisp compiler can only translate a subset of all *Lisp forms. Two conditions must be met before those *Lisp forms that can be *compiled are *compiled:

1. Forms that can be compiled must be made *visible* to the *Lisp compiler.

2. Forms that can be compiled must include complete type declarations.

What do we mean by making code *visible* to the *Lisp compiler? Consider the following piece of *Lisp code.

```
(defun foo (x y)
  (sin!! (+!! (*!! x (+!! y (!! 2))))))
```

If the *Lisp compiler is enabled (as it is by default), it is invoked by calling the Common Lisp compiler on a form or file containing *Lisp code. If the Common Lisp compiler is called on the definition of **foo**, the *Lisp compiler will not compile **foo**. Why not?

Basically, the *Lisp compiler is a very fancy macro expander. This means that the *Lisp compiler is only triggered when the Common Lisp compiler expands a *Lisp macro call. Note that **sin!!**, **+!!**, ***!!**, and **!!** are not *Lisp macros—they are defined as *Lisp functions. In the above code there is nothing to trigger the *Lisp compiler and hence this code is not visible to it.

The obvious question at this point is: "Which *Lisp macros trigger the *Lisp compiler?" Almost all of them, in one manner or another. (Refer to part I, section 2.1.1 for a complete list.) A good general rule is that any *Lisp macro or *defun whose name begins with * will cause its arguments to be compiled by the *Lisp compiler. For example, the statement

```
(*sum (*!! (+!! x y) (!! 2.3)))
```

will be processed by the *Lisp compiler, because the ***sum** triggers the *Lisp compiler.

Since ***defun** itself is a *Lisp macro, one might think that taking the **foo** function and making it a ***defun** would cause it be *compiled:

```
(*defun foo (x y)
  (sin!! (+!! (*!! x (+!! y (!! 2))))))
```

Unfortunately, this is not the case; ***defun** is an exception to our general rule. The *Lisp compiler does not *compile the body of a ***defun**. A particular form in the body of a ***defun** may or may not be *compiled, depending on whether it is visible as defined above. Consider the following ***defun**.

```
(*defun bar (x y)
  (*set x (+!! x (!! 2)))
  (sin!! (+!! (*!! x (+!! y (!! 2)))))
  )
```

The first form of the body of the ***defun** is visible to the *Lisp compiler because ***set** is a *Lisp macro that triggers the *Lisp compiler. The second form is not visible to the *Lisp compiler because **sin!!** is not a *Lisp macro.

Two more exceptions to our general rule about which forms are visible to the *Lisp compiler are ***let** and ***let***. The body of a ***let** or a ***let*** is not necessarily visible to the *Lisp compiler. The initial value forms in these constructs are, however, visible to the *Lisp compiler. For example, given

```
(*let ((temp (!! 3)))
 (+!! temp (*!! temp (!! 2))))
```

the *Lisp compiler will attempt to *compile the binding of the **temp** variable to **(!! 3)**. This is the case because the ***let** macro is semantically equivalent to

```
(*let (temp)
  (*set temp (!! 3))
  (+!! temp (*!! temp (!! 2)))
  )
```

and ***set** triggers the *Lisp compiler. These exceptions are simply current limitations of the *Lisp compiler and may be removed in a future release.

How then do we make the body of **foo** visible to the *Lisp compiler? Here is a trick. Rewrite **foo** as

```
(defun foo (x y)
  (*let ((temp (sin!! (+!! (*!! x (+!! y (!! 2)))))))
    temp
    ))
```

Or as

```
(defun foo (x y)
  (*let (temp)
    (*set temp (sin!! (+!! (*!! x (+!! y (!! 2))))))
    temp
    ))
```

In either case, ***set** triggers the *Lisp compiler. It then attends to all the forms within the ***set**.

The first condition for *compilation has been met. At this point, we know **foo** is visible to the *Lisp compiler; all computation is done inside a macro to which the *Lisp compiler attends. It is possible to make code visible to the *Lisp compiler and still not have it *compile. Notice that there is no tool to report whether a piece of *Lisp code is visible.

Next, we consider how to get the *Lisp compiler to translate visible *Lisp code into Lisp/Paris.

Any code that is visible to the *Lisp compiler but that the *Lisp compiler cannot translate into Lisp/Paris is simply left untranslated. By default, the *Lisp compiler does not report that it has failed to *compile code made visible to it. To receive such reports, set the *Lisp compiler **Warning Level** to **High**. From within the Common Lisp interpreter, with *Lisp loaded, call the function **(compiler-options)**. Alternatively, on a Symbolics Lisp machine, type **Meta-x Set Compiler Options**. (See section 4 in part I for complete instructions.)

Now, set the *Lisp compiler **Warning Level** to **High** and call the Common Lisp compiler on our revised definition of **foo**. The following warning message is displayed:

```
For Function foo
Verbose: While compiling y:
  The expression (*set-1 temp (sin!! (+!! #))) is not compiled
because  +!! does not understand undeclared expressions.
```

```
Verbose: While compiling X:
  The expression (*SET-1 TEMP (SIN!! (+!! #))) is not compiled
because  *!! does not understand undeclared expressions.
  Verbose: While compiling TEMP:
  The expression (*SET-1 TEMP (SIN!! (+!! #))) is not compiled
because  *SET does not understand undeclared expressions.
```

This is not too surprising. We have not satisfied the second condition for *compilation: we have not provided type declarations for the pvars **x**, **y**, and **temp**. Paris is a typed language, which operates on unsigned integers, signed integers and floats. If the *Lisp compiler is not informed of expression types, there is very little translation it can perform.

Let's declare **x**, **y** and **temp**.

```
(defun foo (x y)
  (declare (type (pvar single-float) x y))
  (*let (temp)
    (declare (type (pvar single-float) temp))
    (*set temp (sin!! (+!! (*!! x (+!! y (!! 2))))))
    temp
    ))
```

Attempting to compile, we again receive warning messages. These message indicate that the *Lisp compiler still does not know the types of **x** and **y**— it appears satisfied with the declaration provided for **temp**.

```
For Function FOO
Verbose: While compiling Y:
  The expression (*SET-1 TEMP (SIN!! (+!! #))) is not compiled
because  +!! does not understand undeclared expressions.
Verbose: While compiling X:
  The expression (*SET-1 TEMP (SIN!! (+!! #))) is not compiled
because  *!! does not understand undeclared expressions.
```

Why did this happen? The first **declare** statement is never seen by *Lisp, let alone by the *Lisp compiler. The **declare** form is processed by the Common Lisp compiler because it is part of the **defun** special form and **defuns** are permitted to include **declare**. There is currently no mechanism to force the Common Lisp compiler to pass on to the *Lisp compiler declarations containing pvar information. So, the **declare** form simply gets thrown away. (For more information about the syntax and semantics of **defun** and **declare**, refer to chapters 5 and 9 of *Common Lisp: the Language.*)

How do we inform *Lisp of the types of **x** and **y**? There are three solutions.

In the first solution, we change the **defun** to a ***defun**. The *Lisp ***defun** construct checks for declarations and, if a declaration is a pvar type declaration, it passes that information to the *Lisp compiler.

```
(*defun foo (x y)
  (declare (type (pvar single-float) x y))
  (*let (temp)
    (declare (type (pvar single-float) temp))
    (*set temp (sin!! (+!! (*!! x (+!! y (!! 2))))))
    temp
    ))
```

In the second solution, we use the Common Lisp special form **the** to make type information available to the *Lisp compiler. The *Lisp compiler processes **the** special forms.

```
(defun foo (x y)
  (*let (temp)
    (declare (type (pvar single-float) temp))
    (*set temp
      (sin!! (+!! (*!! (the (pvar single-float) x)
                       (+!! (the (pvar single-float) y)
                            (!! 2))))))
    temp
    ))
```

In the third solution, we use the *Lisp macro ***locally** (new with Release 5.0) to declare the types of **x** and **y**.

```
(defun foo (x y)
  (*locally
    (declare (type (pvar single-float) x y))
    (*let (temp)
      (declare (type (pvar single-float) temp))
      (*set temp (sin!! (+!! (*!! x (+!! y (!! 2))))))
      temp
      )))
```

With the *Lisp compiler **Warning Level** still set to **High**, call the Common Lisp compiler on each of the versions of **foo** to verify that they each do *compile.

We have now successfully transformed a simple,*Lisp program that cannot be *compiled into a *Lisp program that can be *compiled. It has become more complicated, and it has become less general (it will now work only for floating-point arguments), but it will run many times faster than our first version.

If we fully macroexpand the above function and then strip out all but the code pertaining directly to the ***set** statement, the result is the following Lisp/Paris code:

```
(progn (cm:f-add-constant-3-11 (aref temp 1)
         (aref y 1) 2.0 23 8)
       (cm:f-multiply-2-11 (aref temp 1) (aref x 1) 23 8)
       ;; compile to preferred - sin!!.
       (cm:f-sin-2-11 (aref temp 1)
         (aref temp 1) 23 8)
       nil)
```

Notice that, although there are six *Lisp operations in the *set form, the *Lisp compiler reduces these to three Paris instructions.

8 How to UseType Information

The most important concepts related to type specification turn on an understanding of what declaration forms do and do not do. In this section, we will review exactly what **declare**, ***proclaim**, ***locally**, and **the** forms add to a program.

8.1 What Declarations Mean

A declaration, proclamation, or a **the** statement, is a *promise* to either the *Lisp compiler or the Common Lisp compiler. It is not a directive. Type declarations do not persuade a compiler to perform additional type coercion; if promises are broken, errors result.

Consider our old **foo** example:

```
(*defun foo (x y)
  (declare (type (pvar single-float) x y))
  (*let (temp)
(declare (type (pvar single-float) temp))
(*set temp (sin!! (+!! (*!! x (+!! y (!! 2))))))
temp
))
```

Suppose we make the call:

```
(foo (!! 2) (!! 3))
```

This call violates our promise to pass **foo** only pvars of type **single-float-pvar**. Our declarations allowed the *Lisp compiler to translate our **foo** definition into Paris instructions that deal exclusively with floating-point numbers. Passing in integer data has predictable results: the floating-point operators process the integer data, expecting it to be floating-point data, and produce garbage!

At this point, it is appropriate to point out that **declare** forms used with ***let** forms serve two purposes:

1. They make it possible to promise the *Lisp compiler that certain variables always take particular types of data.

2. They make it possible to inform the ***let** macro what type of pvar to allocate for each temporary variable at run time and how much CM memory to reserve for each.

This sort of dual purpose is also served by ***proclaim** forms that precede ***defvar** forms. These serve both to inform the compiler and to inform the run-time state.

This is not a contradiction. While declarations within a ***let** are promises to the *Lisp compiler and nothing more, they are also used as directives to the ***let** macro itself, even though this use is unrelated to *compilation. Declarations are promises to compilers, and they may have additional meanings.

With Release 5.0, the *Lisp compiler is able to generate code that verifies type declaration promises. This capability is enabled by setting the *Lisp compiler **Safety Level** to **3** (for maximum safety).

Using maximum safety, a call to **(foo (!! 2) (!! 3))** results in the following error:

```
Error: #<FIELD-Pvar 52-2 *DEFAULT-VP-SET* (32 16)> was de-
clared to be a (PVAR SINGLE-FLOAT), but isn't.
  Arg 0 (SLC::WHAT): #<FIELD-Pvar 52-2 *DEFAULT-VP-SET* (32
16)>
  Arg 1 (TYPE): (PVAR (DEFINED-FLOAT 23 8))
```

```
s-A, :   Ignore error.
s-B, :  Return to Breakpoint ZMACS in Editor Typeout Window 1
s-C:    Editor Top Level
s-D:    Restart process Zmacs Windows
```

In contrast, at **Safety Level 0** (safety disabled), an unintelligible error message is displayed as shown below.

```
Error: Trying to access off of the end of field 1376256.
The passed field has a length of 2, and the length passed
to this instruction is 32.
CMI::WTL3132-ADD-CONSTANT-3 2
   Arg 0 (CMI::DESTINATION): 1638400
   Arg 1 (CMI::SOURCE): 1376256
   Arg 2 (CMI::CONSTANT-UPPER): 16384
   Arg 3 (CMI::CONSTANT-LOWER): 0
s-A, :  Return to Breakpoint ZMACS in Editor Typeout Window 1
s-B:    Editor Top Level
s-C:    Restart process Zmacs Windows
```

With safety disabled, no error is even signaled in most cases. Setting the *Lisp compiler **Safety Level** to **3** has many debugging benefits beyond yielding decipherable error messages. Until code is fully debugged, it is prudent to always enable maximum *compiler safety. (See part I of the **Lisp Compiler Guide* for a more complete discussion of safety level.)

As another example, if we *compile the expression

```
(*set (the (pvar single-float) x) (!! (the single-float y)))
```

and it turns out that the value of y is not a Common Lisp single-float but a Common Lisp double-float, then we have violated our promise and the code does not work properly. Similarly, if **x** is not a single-float pvar but a double-float pvar (or an **(unsigned-byte 16)** pvar, or anything else), then the code does not work.

Further, code that has successfully run without declarations can fail to run or produce incorrect results once declarations are inserted if those declarations are not true! For instance,

```
(*set x (!! j))
```

will work just fine if x is not declared (it defaults to a general mutable pvar) and if j is 256.

But if x is declared to be of type (pvar (unsigned-byte 8)) and j is 256, then, depending on safety level, the code either errors out or puts the wrong answer into x because 256 cannot be represented in eight bits.

If we write

```
(*set (the (pvar (unsigned-byte 32)) x)
 (the (pvar (unsigned-byte 32)) y))
```

when instead x is only 31 bits long, then we cause some random bit of memory not belonging to x to be overwritten with the most significant bit of y. This is the danger of type declarations: they allow *compilation and they increase performance, but they shift the burden of "doing the right thing" from *Lisp to the programmer.

8.2 What Can Be Declared and How

There are seven basic categories of declarations used to provide the *Lisp compiler with type information.

1. A ***proclaim** form globally declares the type of a pvar and must precede the ***defvar** form that defines that pvar.
2. A temporary pvar may be declared within a ***let** or ***let*** form using a **declare** form.
3. The arguments to *Lisp functions are declared using **declare** forms inside ***defun** forms.
4. The type of pvar returned by a function is globally declared using **function** or **ftype** forms within ***proclaim** forms.
5. The types of scalar variables can be globally declared using ***proclaim**.
6. Scalar variables defined by **let** or **do** may be declared using ***locally**.
7. The arguments of functions not defined with ***defun** may be declared using ***locally**.

Examples of each method of communicating type information to the *Lisp compiler are given below.

Examples:

To create pvars of defined type, use the ***proclaim** macro followed the ***defvar** macro.

```
(*proclaim '(type (pvar single-float) x))
(*proclaim '(type (pvar boolean) bool))
(*proclaim '(type (pvar (unsigned-byte 32)) j))
(*proclaim '(type (pvar (complex single-float)) c))
(*defvar x (!! 2.0))
(*defvar bool nil!!)
(*defvar j (!! 0))
(*defvar c (!! #c(1.0 1.0)))
```

To declare the types of pvars defined by ***let** and to declare the types of arguments to ***defun** functions, use **declare**.

```
(*let ((x (!! 2.0)) (bool nil!!) (j (!! 0)) (c (!! #c(1.0
1.0))))
  (declare (type (pvar single-float) x))
  (declare (type (pvar boolean) bool))
  (declare (type (pvar (unsigned-byte 32)) j))
  (declare (type (pvar (complex single-float)) c))
  <this is the body of the *let>
  )
```

```
(*defun (x bool j c)
  (declare (type (pvar single-float) x))
  (declare (type (pvar boolean) bool))
  (declare (type (pvar (unsigned-byte 32)) j))
  (declare (type (pvar (complex single-float)) c))
  <this is the body of the *defun>
  )
```

To declare the type of a pvar returned by a function, use **ftype** within a ***proclaim** macro. For instance, the **foo** function defined at the beginning of this tutorial returns a single-float pvar. This can be declared as follows:

```
(*proclaim '(ftype (function (t t) (pvar single-float)) foo))
```

The above form informs the *Lisp compiler that **foo** is a function that accepts two arguments whose type we don't care about and returns a pvar of type single-float.

These function declarations are extremely useful. For example,

```
(*set (the (pvar single-float) x)
  (+!! (foo (!! 1.2) (!! 1.3)) (!! 1.0)))
```

Could not be *compiled if the return value of **foo** had not been declared.

If we attempt to compile the above ***set** expression without first proclaiming the return value of foo, the warning message shown below is issued.

```
Warning: Verbose: While compiling (FOO (!! 1.2) (!! 1.3)):
 The expression (*SET-1 (THE (PVAR SINGLE-FLOAT) X)
 (+!! (FOO # #) (!! 1.0))) is not compiled because +!! does
 not understand undeclared expressions.
```

It is possible to tell the *Lisp compiler about the types of Common Lisp variables by using the ***proclaim** macro. For instance the proclamation

```
(*proclaim '(type single-float scalar-x))
```

is visible to the *Lisp compiler and informs it of the type of **scalar-x**. Given this proclamation, the expression

```
(*set (the (pvar single-float) x) (!! scalar-x))
```

can be *compiled. If the type of **scalar-x** is *not* proclaimed, the following warning is issued.

```
Warning: Verbose: While compiling scalar-x: The expression
(*set-1 (the (pvar single-float) x) (!! scalar-x)) is not com-
piled because !! doesn't understand undeclared expressions.
```

The *Lisp compiler does not know what type of value to expect to be passed to !! in the **(!! scalar-x)** form within the ***set**.

As another example consider

```
(dotimes (j 10)
 (*set (the (pvar (unsigned-byte 32)) x) (!! j)))
```

This does not *compile properly. Remember that the *Lisp compiler is not triggered until the ***set** is invoked. The **dotimes** is not visible to the *Lisp compiler, which therefore is never informed that **j** must be an integer. This statement must be rendered as

```
(dotimes (j 10)
 (*set (the (pvar (unsigned-byte 32)) x) (!! (the fixnum j))))
```

Often the trick to making *Lisp code *compile, is putting **the** forms into **!!** expressions as in,

```
(!! (the fixnum j))
```

and

```
(!! (the single-float x))
```

This can be a chore. The simple macros below can help speed this process.

```
(defmacro !!tf (x) `(!! (the fixnum ,x)))

(defmacro !!tsf (x) `(!! (the single-float ,x)))

(defmacro !!tdf (x) `(!! (the double-float ,x)))

(defmacro !!tscf (x) `(!! (the (complex single-float) ,x)))

(defmacro !!tdcf (x) `(!! (the (complex double-float) ,x)))

(defmacro !!tsch (x) `(!! (the string-char ,x)))

(defmacro !!tch (x) `(!! (the character ,x)))
```

Using these macros can greatly decrease the amount of typing one needs to do. For instance:

```
(!! (the fixnum scalar)) => (!!tf scalar)

(!! (the (complex single-float) z)) => (!!tscf z)
```

Feel free to copy and use these macros.

Another way to avoid having to repeatedly type **the** forms is to add ***locally** forms instead of **the** forms to code that uses variables more than once inside forms visible to the *Lisp compiler. For instance,

```
(defun xyzzy (foo bar)
  (dotimes (j 100)
    (*set (the single-float-pvar foo) (!! (the fixnum j)))
    (*set (the single-float-pvar bar)
      (*!! (the single-float-pvar foo) (!! (the fixnum j))))
    ))
```

could be rendered more succinctly using ***locally** as shown below.

```
(defun xyzzsy (foo bar)
  (dotimes (j 100)
    (*locally
       (declare (type fixnum j))
       (declare (type single-float-pvar foo bar))
       (*set foo (!! j))
       (*set bar (*!! foo (!! j)))
       )))
```

A final example illustrates the effective use of **the** to get a complicated expression to *compile. The ***set** statement below,

```
(defun hard-to-*compile (y)
  (*set (the (pvar (unsigned-byte 32)) x)
    (cond
      ((eq y 5) (!! 3))
      ((eq y 1000) (!! 2500))
      (t (!! 0))
      )))
```

if processed by the *Lisp compiler, yields this warning message:

```
Warning: IF special form
(IF (EQ Y 5) (PROGN (!! 3)) (IF (EQ Y 1000) (PROGN #) (IF T # NIL)))
not yet implemented.
```

A version that will pass the *Lisp compiler without warnings is written as:

```
(defun was-hard-to-*compile (y)
(*set (the (pvar (unsigned-byte 32)) x)
      (the (pvar (unsigned-byte *))
           (cond
             ((eq y 5) (!! 3))
             ((eq y 1000) (!! 2500))
             (t (!! 0))
       ))
  ))
```

Why is this the case? Given the first version, the *Lisp compiler tries to *compile the source for the ***set**, which is the **cond** form. The *Lisp compiler does have information about **cond**, per se, but notices that it is a Common Lisp macro. Macroexpanding the **cond** produces an **if** form, which is mentioned in the error message. The *Lisp compiler cannot currently handle Common Lisp special forms or Common Lisp functions. If it encounters any of these in the middle of *compilation, it will abandon its attempt to *compile the enclosing form.

In the second version, we explicitly inform the *Lisp compiler that the **cond** form returns a mutable unsigned-byte pvar. Given this information, the *Lisp compiler can generate code that copies the pvar returned by the **cond** statement into the destination pvar **x**.

Using mutable pvar declarations (type declarations that use * as length specifiers) adds flexibility to *compiled code. This is covered in some detail in the next section.

9 How to Write General *Lisp Code that *Compiles

Operations that accept a variety of inputs contribute generality and modularity to programs. Given the *Lisp compiler's requirement for type information, how can we write functions that *compile without sacrificing the flexibility inherent in a functional language? In this section, we explore three techniques for writing semi-generic operations that will, nonetheless, *compile.

9.1 Generalizing Functions Based on Argument Types

Let us consider again the function **foo**:

```
(*defun foo (x y)
  (declare (type (pvar single-float) x y))
  (*let (temp)
    (declare (type (pvar single-float) temp))
    (*set temp (sin!! (+!! (*!! x (+!! y (!! 2))))))
    temp
    ))
```

Remember that in order to get **foo** to *compile, we had to restrict its inputs to single-float pvars. The problem with this solution is that, were it not for the input type restrictions, this function definition might be usefully employed with other types of numeric data, such as floating-point, complex, or integer pvars. How can we make **foo** accept these other types of input arguments?

To answer this question, we try relaxing our assumptions about **foo**'s arguments in a variety of ways and test alternative assumptions one at a time.

First, suppose we want **foo** to work with any kind of floating-point inputs rather than simply with single-precision floating-point pvars. We can write:

```
(*defun foo (x y)
  (declare (type (pvar (defined-float * *)) x y))
  (*let (temp)
    (declare (type (pvar single-float) temp))
    (*set temp (coerce!! (sin!! (+!! (*!! x (+!! y (!! 2)))))
                         '(pvar single-float)))
    temp
    ))
```

The declaration **(pvar (defined-float * *))** informs the *Lisp compiler that x and y can be of any (legal) floating-point size. In the ***set** form, x and y are processed in whatever floating-point format they initially bear and the result is coerced—with a possible loss of precision—into a single-float pvar.

Next suppose we want **foo** to work with either single-float arguments or integer arguments. To test the type of a pvar we can use the Common Lisp function **typep**.

```
(typep (!! 3.0) '(pvar single-float)) => t
(typep (!! 3.0) '(pvar (unsigned-byte *))) => nil
```

Using **typep**, we can conditionalize **foo** based on input type.

```
(*defun foo (x y)
  (*let (temp xtemp ytemp)
    (declare (type (pvar single-float) temp xtemp ytemp))
    (cond
      ((typep x '(pvar single-float))
       (*set xtemp (the single-float-pvar x)))
      ((typep x '(pvar (unsigned-byte *)))
       (*set xtemp (the (pvar (unsigned-byte *)) x)))
      ((typep x '(pvar (signed-byte *)))
       (*set xtemp (the (pvar (signed-byte *)) x)))
      )
    (cond
      ((typep y '(pvar single-float))
       (*set ytemp (the single-float-pvar y)))
      ((typep y '(pvar (unsigned-byte *)))
       (*set ytemp (the (pvar (unsigned-byte *)) y)))
      ((typep y '(pvar (signed-byte *)))
       (*set ytemp (the (pvar (signed-byte *)) y)))
      )
    (*set temp (sin!! (+!! (*!! xtemp (+!! ytemp (!! 2))))))
    temp
    ))
```

This rendering has the effect of converting the input arguments into single-float pvars and then performing the body of the function.

Now, suppose we want a version of **foo** that works with single-floats, double-floats, complex single-floats, or complex double-floats and suppose we want the result of **foo** to be of the same type as its inputs. For simplicity, we assume that **x** and **y** are always of the same type.

```
(*defun foo (x y)
  (macrolet
    ((foo-body (type x y)
       `(*let (temp)
          (declare (type ,type temp))
          (*set temp
                (sin!! (+!! (*!! (the ,type ,x)
                                 (+!! (the ,type ,y) (!! 2)))))
          temp
          )))
    (cond
((typep x '(pvar single-float))
 (foo-body (pvar single-float) x y))
((typep x '(pvar double-float))
 (foo-body (pvar double-float) x y))
((typep x '(pvar (complex single-float)))
 (foo-body (pvar (complex single-float)) x y))
((typep x '(pvar (complex double-float)))
 (foo-body (pvar (complex double-float)) x y))
)))
```

Often what we really want, especially when using a CM-2 system with single-precision floating-point hardware, is a very fast version for single-float arguments, and simply something that works otherwise. If this is the case, we can redefine **foo** as follows.

```
(*defun foo (x y)
  (if (and (typep x '(pvar single-float))
           (typep y '(pvar single-float)))
      (*let (temp)
        (declare (type (pvar single-float) temp))
        (*set temp
              (sin!! (+!! (*!! (the (pvar single-float) x)
                               (+!! (the (pvar single-float) y)
                                    (!! 2))))))
       temp
       )
      (compiler-let ((*compilep* nil))
        (sin!! (+!! (*!! x (+!! y (!! 2)))))
        )))
```

This rendition makes the *Lisp compiler generate a very fast version of the code for single-float inputs while letting the *Lisp interpreter handle all other cases. The **(compiler-let ((*compilep* nil))** form turns off the *Lisp compiler for the body of the **compiler-let**. A useful macro more expressive of this concept is:

```
(defmacro *nocompile (&body body)
  `(compiler-let ((*compilep* nil)) ,@body))
```

Using this ***nocompile** macro, the **compiler-let** expression may be rendered:

```
(*nocompile (sin!! (+!! (*!! x (+!! y (!! 2))))))
```

Similarly, it is trivial to define a ***compile** macro to turn on the *Lisp compiler.

```
(defmacro *compile
  ((&key (safety 1) (warning-level :high)) &body body)
   `(compiler-let ((*compilep* t) (*safety* ,safety)
                   (*warning-level* ,warning-level))
    ,.body))
```

With these two macros, we can write operations that switch between code intended for optimization by the *Lisp compiler and normal evaluation by the *Lisp interpreter. Once code is debugged, we may want to *compile it without error checking; this will speed up the execution of *compiled code. Here is a macro to turn on the *Lisp compiler with no safety.

```
(defmacro *compile-blindly (&body body)
  `(compiler-let
     ((*compilep* t) (*safety* 0) (*warning-level* :high))
       ,.body))
```

9.2 Generalizing Functions Based on the Lengths of Arguments

As a slightly different example of generalizing *Lisp code, let's try to define a function that takes a signed pvar of unspecified size and adds 17 to it. Here is the basic function:

```
(defun bar (x) (*set x (+!! x (!! 17))))
```

To *compile it, we might write

```
(*defun bar (x)
  (declare (type (pvar (signed-byte *)) x))
  (*set x (+!! x (!! 17)))
  )
```

Unfortunately, the form **(declare (type (pvar (signed-byte *)) x))** declares **x** as a mutable signed-byte pvar. This promises not only that is **x** is of unspecified size but that the size of **x** can actually *change.* (For more information on mutable pvars, see chapter 8 of the *Supplement to the *Lisp Reference Manual.*) As long as **x** is not modified (i.e, as long as it is used only as an rvalue) this does not matter. If we modify **x**, as in the ***set** in the above code, the semantics of the declaration dictate that the size of **x** may change. While the *Lisp Compiler can generate code that allows pvars to change size as required, this is not necessarily efficient.

Assuming we know that the result of adding 17 to **x** will not overflow, it is not necessary to direct the *Lisp compiler to allow **x** to change size. In this case, the proper way to define the **bar** function is:

```
(*defun bar (x)
  (declare (type (pvar (signed-byte (pvar-length x))) x))
  (*set x (mod!! x (!! 10)))
  )
```

Here an expression is substituted for the length argument in the pvar **type** declaration. This arcane declaration informs the *Lisp compiler that **x** is a signed-byte pvar that is "as long as it is."

Unlike Common Lisp declarations, virtually any arbitrary expression may be used for the length arguments to pvar type declarations. The **baz** definition below illustrates.

```
(defun baz (y)
  (*let (temp)
    (declare (type (pvar
      (unsigned-byte (min (* (pvar-length y) 2) 32)))
     temp))
    ))
```

This sort of parameterization is another powerful tool in generalizing function definitions.

9.3 Generalizing Simple Functions by Making Them Macros

A final technique for generalizing a simple function to simply turn it into a macro. Let's go back to our original definition of **foo**:

```
(defun foo (x y) (sin!! (+!! (*!! x (+!! y (!! 2))))))
```

As a macro, we can write **foo** as:

```
(defmacro foo (x y) `(sin!! (+!! (*!! ,x (+!! ,y (!! 2))))))
```

As long as the types of x and y are known to the *Lisp compiler inside of the functions that call the **foo** macro, we win.

```
(*defun f1 (x y)
  (declare (type (pvar single-float) x))
  (declare (type (pvar (unsigned-byte 8)) y))
  (*set x (foo x y))
  x )


(*defun f2 (x y)
  (declare (type (pvar (complex single-float)) x))
  (declare (type (pvar double-float) y))
  (*set x (foo x y))
  x )
```

Here, we let the *Lisp compiler determine how to *compile the **(sin!!** (+!! (*!! x (+!! y (!! 2))))) expression efficiently each time the **foo** macro is called in a program. While this kind of trick can waste code space, the benefits of code flexibility and reduced programming time often outweigh this cost.

In summary, with a little care, it is possible to write *Lisp code that is both flexible and efficient. Conditionalizing with the **typep** function, forcing type-restricted return values with the **coerce!!** function, using pvar declarations with indefinite length specifiers, and using macros are all viable means to this end.

Index

Index