

# Compiling

This document contains the following sections:

[1.0 Lisp images without a compiler](#)

[2.0 Compile-file](#)

[2.1 Persistence of defining forms encountered by compile-file](#)

[3.0 Compiler warnings: undefined function, inconsistent name usage, unreachable code](#)

[4.0 EOF encountered error](#)

[5.0 Variables which control information printed by compile-file](#)

[6.0 File types](#)

[7.0 Declarations and optimizations](#)

[7.1 Examining the current settings and their effect](#)

[7.2 Declared fixnums example](#)

[7.3 Argument count checking example](#)

[7.4 Argument type for a specialized function example](#)

[7.5 Bound symbol example](#)

[7.6 Tail merging discussion](#)

[7.7 Changing compiler optimization switch settings](#)

[7.8 Switch can be t or nil meaning always on or always off](#)

[8.0 Pointers for choosing speed and safety values](#)

[8.1 Optimizing for fast floating-point operations](#)

[8.2 Supported operations](#)

[9.0 Help with declarations: the :explain declaration](#)

[10.0 Other declarations and optimizations](#)

[10.1 Inline declarations ignored](#)

[10.2 Defstruct accessors](#)

[10.3 Stack consing, avoiding consing using apply, and stack allocation](#)

[10.4 Adding typep-transformers](#)

[10.5 Compiler handling of top-level forms](#)

The Common Lisp standard requires that a Common Lisp system have a compiler and a loader. Since running interpreted code in any Common Lisp system is slow compared to running compiled code, most users will want to compile functions after trying them out in interpreted mode. A compiled function will run many times faster than its interpreted counterpart.

There are several implementation-dependent facets to a Common Lisp compiler. These include the naming of source and object files, how declarations are handled, and how floating-point and other

optimizations are performed. These issues are discussed in this document.

---



---

## 1.0 Lisp images without a compiler

Building an image with **build-lisp-image** (see *building-images.htm*) with *include-compiler* `nil` (or *include-compiler* `true` and *discard-compiler* also `true`) results in a compilerless Lisp image.

Standard runtime images (see *runtime.htm*) may not have the compiler included. In such images, it is an error to call **compile** or **compile-file** directly or indirectly. The value of the `*compiler-not-available-warning*` (discussed further below) variable is ignored in standard runtime images and thus does not affect whether or not an error is signaled. The condition signaled is `runtime-restriction`.

The remainder of this section discusses images which are not standard runtime images.

It is possible to load the compiler into a compilerless image. You do so by evaluating `(require :compiler)`. However, it may be difficult or impossible to compile everything that would have been compiled had the compiler been present earlier (difficult or impossible because you do not know what the things are). Therefore, parts of Lisp will run slower than necessary after the compiler is loaded with no space saving (since you loaded the compiler). In short, if you intend to use the compiler, start with an image that includes it!

But note that you can build an image with the compiler included during the build and discarded after the build completes. This might be very useful when creating an application for distribution when you are not licensed to distribute an image with the compiler. If you specify `:include-compiler t` and `:discard-compiler t` in a call to **build-lisp-image** or **generate-application** (which accepts `build-lisp-image` arguments), the compiler will be present while the image is built and discarded at the end. Thus, things like applying advice (see *fwrappers-and-advice.htm*) can be done and compiled during the build. (Note that the new `fwrapper` facility, described in the same document, works with objects that can be pre-compiled.)

If you use a compilerless Lisp, the system will (in the default) print a warning every time you or the system tries to call **compile**. For example, consider the following script from a compilerless Lisp image:

```
USER(1): (defun foo nil nil)
FOO
USER(2): (compile 'foo)
Warning: Compiler not loaded -- function left uncompiled
NIL
```

```
NIL
NIL
USER( 3 ) :
```

The text of the warning changes according to what the system was unable to compile. We emphasized that **compile** can be called by the system since some operations, including calling **advise** under certain circumstances, calling the (obsolete) function **defforeign** (but not calling its replacement macro **def-foreign-call**) in a *fasl* file, and doing non-standard method combinations in CLOS or MOP usage, do cause the compiler to be called and the warning to be printed. It is possible to suppress these warnings, either globally or locally. The variable `*compiler-not-available-warning*` controls whether the warning is printed or not.

We do not recommend setting `*compiler-not-available-warning*` to `nil` because interpreted code will run significantly more slowly than compiled and users should know when the compiler was called so they can expect degraded performance.

It is possible to suppress the warning locally. The warnings actually signal the condition `compiler-not-available-warning`. Thus, it is possible to suppress these warnings by wrapping the following code around sections likely to produce such unwanted warnings:

```
(handler-bind ((excl:compiler-not-available-warning
  #'(lambda (c) (declare (ignore c)) (muffle-warning))))
<code-which--may-directly-or-indirectly-calls-compile>)
```

Note again that it is possible to build an image with the compiler included during the build and discarded at the end. Things processed during the build will be processed with the compiler but the final image will not have it (and thus suitable for delivery when the license does not permit distribution of images containing the compiler, such as standard runtime images, see *runtime.htm*). See the description of the *include-compiler* and *discard-compiler* arguments to **build-lisp-image** in *building-images.htm*.

## 2.0 Compile-file

The Common Lisp function **compile-file** is used to compile a file containing Common Lisp source code. We describe that function here to point out some Allegro-CL-specific arguments. The link above is to the ANSI description.

```
cl:compile-file
```

**Arguments:** *input-filename* &key *output-file* *xref* *verbose* *print* *fasl-circle*

This function returns three values. The first value is the truename of the output file, or `nil` if the file could not be created. The second value is `nil` if no compiler diagnostics were issued, and `true` otherwise. The third value is `nil` if no compiler diagnostics other than style warnings were issued. If the third returned value is `true`, there were serious compiler diagnostics issued or the other conditions of type `error` or type `warning` were signalled during compilation.

This function will compile the Common Lisp forms in *input-filename*. *input-filename* must be a pathname, a string, or a stream (earlier versions of Allegro CL accepted a symbol as the filename argument but this is now explicitly forbidden by the ANSI standard).

*input-filename* is merged with `*default-pathname-defaults*`. If *input-filename* name has no type, then **compile-file** uses the types in the list `*source-file-types*`. The name of the output file may be specified by the *output-file* keyword argument. The default type of the output file is specified by the variable `*fasl-default-type*`.

The output file will have a text header (readable, e.g., by the Unix command **head**) which provides information such as the time of creation and the user who created the file.

The *xref* keyword argument is a boolean that controls whether cross-referencing information is stored in the output file. The default value of the *xref* keyword argument is the value of `*record-xref-info*`.

The *verbose* keyword argument is a boolean that controls printing of the filename that is currently being compiled. The default value of the *verbose* keyword argument is the value of `*compile-verbose*`.

The *print* keyword argument is a boolean that controls whether **compile-file** prints the top-level form currently being compiled. The default value of the *print* keyword argument is the value of `*compile-print*`.

The *fasl-circle* keyword argument is an Allegro CL extension (i.e. may not be supported in versions of Common Lisp other than Allegro CL). If the value of this argument is `nil`, there are two effects:

- **make-load-form** processing is not done. CLOS instances can be written to *fasl* files only if **make-load-form** processing is done and structures are dumped as tagged vectors (instead of using **make-load-form** processing).
- The requirement that the compiler is required to preserve **eql**-ness within a file compiled by **compile-file** -- will not be met.

On the other hand, compilation speed is typically faster when this argument is `nil`. This argument

defaults to the value of `*fasl-circle-default*`.

If a definition of an object (with a defining macro like **defpackage**, **defmacro**, **defstruct**, etc.) within a file being compiled with **compile-file** persists in Lisp after the compilation is complete, the definition is said to be persistent. The persistence behavior prior to release 7.0 was erratic, with some defining forms persisting and others not. The situation in 7.0 is fully described in [Section 2.1 Persistence of defining forms encountered by compile-file](#). The short story is, absent explicit direction (such as with an **eval-when**), the only persistent defining form is **defpackage**.

The **:cf** top-level command invokes **compile-file**. **:cf** does not accept keyword arguments but does accept multiple input-filename arguments. **:cf** reads its arguments as strings, so you should not surround input-filenames with double quotes. Thus

```
:cf foo.cl
```

is equivalent to

```
(compile-file "foo.cl")
```

Certain versions of Allegro CL distributed as samples have **compile-file** disabled. In such versions, the function **load-compiled**, whose arguments are similar to **compile-file**, will compile a file and load it but not write out a *fasl* (compiled) file.

## 2.1 Persistence of defining forms encountered by compile-file

Suppose in a running Lisp, the symbols `foo` and `foom` do not name any objects, indeed do not even exist in any package. Consider the following file named *foo.cl*:

```
;; file foo.cl begin
(in-package :user)

(defpackage :foo (:use :cl :excl))
(defmacro foom (x) x)
(defun foo (y) (cons y y))
(defclass foo () ())
(defvar foo nil)

;; file foo.cl end
```

We now compile the file using **compile-file**:

```
(compile-file "foo.cl")
```

After the file compilation completes, in the running Lisp, does `foo` now name a package, a function, a class, and a global variable? Does `foom` name a macro?

The answer is, only the package is now defined in the Lisp:

```
cl-user(16): (describe 'foo)
foo is a symbol.
  It is unbound.
  It is internal in the common-lisp-user package.
cl-user(17): (find-package :foo)
#<The foo package>
cl-user(18): (describe 'foom)
foom is a symbol.
  It is unbound.
  It is internal in the common-lisp-user package.
cl-user(19):
```

The ANSI spec does not say whether defining forms persist after a `compile-file` (i.e. the definition should be available in the running Lisp when compilation is complete). The behavior of Allegro CL in releases prior to 7.0 was uneven, with some defining forms persisting and others not. Starting in release 7.0, the behavior is consistent: only **defpackage** forms persist.

We believe this new consistency will make it easier to know what the best compilation strategy is, particularly now that we support compiler environments (see *environments.htm*). Because it is easy to ensure definitions persist by using **eval-when**'s, we believe that the extra programming burden is slight.

Here is a list of defining macros. Those marked with (\*) persisted in earlier releases but no longer persist. More complicated situations are also noted:

- **declaim** (\*)
- **defclass**
- **defconstant**
- **define-compiler-macro**
- **define-condition**
- **define-modify-macro**
- **define-setf-expander** (\*)
- **defmacro**
- **defpackage** (always was and is now persistent)

- **defparameter** (special declaration persisted, no longer does)
- **defsetf** (\*)
- **defstruct**
- **deftype**
- **defvar** (special declaration persisted, no longer does)
- **define-symbol-macro** (\*)

If you wish a defining form to persist, wrap it in an appropriate **eval-when** form, such as:

```
(eval-when (compile load eval) (defmacro foo ...))
```

## Some examples

Definitions do not persist (absent **eval-when**'s) from one file to another within a compilation unit:

```
;; file foo1.cl begin
(in-package :user)
```

```
(defvar *foo1* nil)
```

```
;; file foo1.cl end
```

```
;; file foo2.cl begin
(in-package :user)
```

```
(defvar *foo2* nil)
```

```
(defun bar ()
```

```
  *foo1*)
```

```
;; file foo2.cl end
```

```
cl-user(1): :cf foo1
```

```
;;; Compiling file foo1.cl
```

```
;;; Writing fasl file foo1.fasl
```

```
;;; Fasl write complete
```

```
cl-user(2): :cf foo2
```

```
;;; Compiling file foo2.cl
```

```
; While compiling bar:
```

```
Warning: Free reference to undeclared variable *foo1* assumed special.
```

```
;;; Writing fasl file foo2.fasl
```

```
;;; Fasl write complete
```

```
cl-user(3): (with-compilation-unit ())
```

```

        (compile-file "foo1.cl")
        (compile-file "foo2.cl"))
;;; Compiling file foo1.cl
;;; Writing fasl file foo1.fasl
;;; Fasl write complete
;;; Compiling file foo2.cl
; While compiling bar:
Warning: Free reference to undeclared variable *foo1* assumed special.
;;; Writing fasl file foo2.fasl
;;; Fasl write complete
#p"foo2.fasl"
t
t
cl-user(4):

```

---



---

### 3.0 Compiler warnings: undefined function, inconsistent name usage, unreachable code

The Allegro CL compiler signals warnings when it detects potential difficulties with the code being compiled. We describe some of these warnings here. These are not errors and the compilation will complete and compiled code will be produced. The warnings simply flag potential problems, perhaps uncovering a coding error which will cause a program problem even though the code compiles.

The following condition classes are among those that might be signaled:

- `compiler-inconsistent-name-usage-warning`: signaled whenever variable or tag names are used in a manner inconsistent with their possibly intended usage, either unused (and for variables, not declared ignore or ignorable), or, again for variables, declared ignore but used.
- `compiler-unreachable-code-warning`: signaled whenever a **cond** clause or a **case** clause cannot be reached, presumably because of the presence of a `t` or `otherwise` clause before the unreachable clause.
- `compiler-undefined-functions-called-warning`: signaled when the compiler notices that there are calls to not-yet-defined functions in a function being compiled. This will not be signaled when the function is defined in the same file or is defined within the body of **with-compilation-unit**.
- `compiler-no-in-package-warning`: this warning is no longer used. Allegro Presto required files to have an **in-package** form but Allegro Presto is no longer supported. See *The Allegro Presto facility has been removed in loading.htm* for further information.

---

---

## 4.0 EOF encountered error

If you try to compile a file that contains an incomplete form (because of, e.g., a missing closing parenthesis), **compile-file** signals an error with condition `end-of-file`. Consider the following file *missing-paren.cl*:

```
(defun foo nil nil)
(defun bar (a b) (+ a b)
```

The closing parenthesis is missing from the definition of **bar**. When Lisp tries to compile this file, it signals an error:

```
USER(3): :cf missing-paren.cl
; --- Compiling file /net/rubix/usr/tech/dm/missing-paren.cl ---
; Compiling FOO
Error: eof encountered on stream"
#<EXCL::CHARACTER-INPUT-FILE-STREAM
#p"/net/rubix/usr/tech/dm/missing-paren.cl" pos 45 @ #xa99c12>
starting at position 20.
[condition type: END-OF-FILE]
```

```
Restart actions (select using :continue):
0: retry the compilation of /net/rubix/usr/tech/dm/missing-paren.cl
1: continue compiling /net/rubix/usr/tech/dm/missing-paren.cl but
generate no
output file
[1] USER(4):
```

Note the line:

```
starting at position 20.
```

That indicates that the incomplete form starts at position 20 in the file. Opening the file in Emacs and entering the command `C-u 20 C-f` (assuming standard keybindings, that means move forward 20 characters) should bring you to the beginning of the incomplete form.

---

---

## 5.0 Variables which control information printed by compile-file

The variables `*compile-print*` and `*compile-verbose*` provide the defaults for the *print* and *verbose* arguments to **compile-file**. Those arguments control how much information **compile-file** will print out. These variables are a standard part of common lisp.

---

## 6.0 File types

The default source-file type in Allegro CL is *cl*. The default compiled-file type in Allegro CL is *fasl*, which is a mnemonic for *fast-loadable* file. The default file types may be changed to suit your preferences.

The variable `*source-file-types*` is a list of pathname types of Common Lisp source files. The initial value of this variable is the list

```
("cl" "lisp" "lsp" nil)
```

This means that if no file type is specified for the argument of **compile-file** (or the top-level command **cf**), files with types *cl*, *lisp*, *lsp*, and no type will be looked for. For example

```
(compile-file "foo")
```

will cause the compiler to first look for the file *foo.cl*, then for *foo.lisp*, the *foo.lsp*, and finally *foo*. Users may want to change the value of `*source-file-types*`. Some prefer not to allow files with no type to be looked at, since this prevents Lisp from trying to compile inappropriate files, or even a directory. Evaluating the form

```
(setq sys:*source-file-types* '("cl" "lisp" "lsp"))
```

will cause the compiler to look for files with file type *cl*, *lisp* and *lsp*. Then

```
(compile-file "foo")
```

will look for *foo.cl*, *foo.lisp*, and *foo.lsp* but not *foo*. The first file found will be the one compiled.

If you change `*source-file-types*`, you may also wish to change `*load-search-list*` and `*require-search-list*` so that the functions `load` and `require` will look for files with the desired file types as well. See *Search lists* and its subsections in *loading.htm* for a description of these variables.

When a file is compiled, a new file containing the compiled Lisp code is created. This file will have the type specified by the variable `*fasl-default-type*`. The initial value of this variable is "fasl". You may change its value to any string you wish. If you change the value of this variable, you should also modify the load and require search lists so those functions will find the correct files. Throughout the documentation, files containing compiled Lisp code are called fasl files and are assumed to have a file type of *fasl*. You should understand that *fasl* really denotes the value of `*fasl-default-type*`.

---

## 7.0 Declarations and optimizations

Compiling Lisp code involves many compromises. Achieving very high speed generally involves sacrificing graceful recovery from errors and even the detection of errors. The latter may cause serious problems such as wrong answers or errors that result from the propagation of the original undiscovered error. On the other hand, interpreted code is very easy to debug but may be too slow for practical applications. Fortunately, most program needs can be satisfied on some middle ground. The software development cycle generally begins with interpreted code and ends with well-optimized code. Progressing through this cycle, higher speed is achieved without significant loss of confidence because of the increase in the correctness and robustness of the Lisp code as development proceeds. (It is important to note that optimizations do not affect the behavior of correct code with expected inputs. However, it is not always easy to prove that the inputs will always be what they are expected to be or that a complex program is indeed `correct'.) This section provides enough information so that a programmer may make intelligent decisions about performance compromises. [Section 8.0 Pointers for choosing speed and safety values](#) further discusses the issue.

Among specific trade-offs in compiling code are the verification of the number and types of arguments passed to functions, providing adequate information for debugging, and including code to detect interrupts. The Allegro CL compiler has been designed to be highly parameterized. Optimizations performed by the compiler may be controlled to a significant degree by the user. The compiler of course supports the primitive Common Lisp safety, space, speed, and debug optimization declarations. (Allegro CL accepts integer values between 0 and 3 inclusive, higher values representing greater importance to the corresponding aspect of code generation.) More significantly, Allegro CL provides a number of optimization switches that control specific aspects of code generation. These switches are in the form of variables that are bound to functions of the four optimization parameters safety, space, speed, and debug.

The initial values of the optimization qualities are set when Allegro CL is built, controlled by the **build-lisp-image** arguments `:opt-safety`, `:opt-space`, `:opt-speed`, and `:opt-debug` (see *Arguments to build-lisp-image 2: defaults not inherited from the running image* in *building-images.htm*).

The default values for safety, space, and speed in an image created by **build-lisp-image** and in prebuilt

images supplied in the Allegro CL distribution is 1. The default value for debug is 2.

You can set the values of the four qualities in various ways. One way is globally, with **proclaim** as follows:

```
(proclaim '(optimize (safety n1)
                    (space n2)
                    (speed n3)
                    (debug n4)))
```

where *n1*, *n2*, *n3*, and *n4* are integers from 0 to 3. The values may also be set with the top-level command **:optimize**. This command also displays the current values of the qualities (there is no portable way to access those values in Common Lisp).

---

## 7.1 Examining the current settings and their effect

The function **explain-compiler-settings** prints the value that will be returned by each compiler switch if called with specific settings of the optimization qualities. Called with no arguments, it describes the behavior with the current settings. It takes keyword arguments named by the optimization qualities (`safety`, `space`, `speed`, and `debug`). Values specified for those arguments causes information on the values of compiler switches using the specified values (and the current values, where unspecified) to be printed.

The variables discussed in the remainder of this section specify what the various settings of safety, space, speed, and debug do. Their values are functions that return `t` or `nil` for the given settings of safety, space, speed, and debug. You may change the definitions by binding (or setting) the variables to new functions.

You can also set (or bind) a variable to `t` or `nil` and this will be interpreted as a function that always returns `t` or `nil` (respectively) meaning the associated switch is always on (`t`) or off (`nil`). Any new function used should accept four arguments. The system calls the function with the (lexically) current values of safety, space, speed and debug.

Following the table describing the switches, we give examples of erroneous code run interpreted (which produces the same error behavior as running with the switches at their safest setting) and run compiled with the switches at their least safe setting. What you will notice in the latter case is that erroneous code either results in a less informative error or perhaps in a wrong answer but no error. These examples are not intended to deter you from using compiler optimizations. Rather, we want to make you aware of the dangers of less safe code and show what error messages to expect when the switches are at their high

speed settings. All switches are named by symbols in the `compiler` package.

Switch	Description	Initial value true when:
<code>compile-format-strings-switch</code>	<p>When true, a format string (the second required argument to <code>format</code>) is converted to a tree structure at compile time. If <code>nil</code>, the conversion is done at run time, resulting in slower printing but smaller code.</p>	<p>True when speed is greater than space.</p>
<code>declared-fixnums-remain-fixnums-switch</code>	<p>If true, compiler will generate code that assumes the sum and difference of declared fixnums are fixnums.</p> <p><b>Warning:</b> if this switch returns true during compilation but the sum or difference of two declared fixnums is not a fixnum, the compiled code will silently produce erroneous results.</p> <p>See <a href="#">Section 7.2 Declared fixnums example</a> below.</p>	<p>True only if speed is 3 and safety is 0.</p>
<code>generate-inline-call-tests-switch</code>	<p><b>This switch is ignored in release 5.0.1 and later.</b> It was used in earlier releases to optimize certain <b>flavors: send</b> macro calls but that is now done differently.</p>	<p>(Not applicable.)</p>

<p><code>generate-interrupt-checks-switch</code></p>	<p>If true, code is added at the beginning of the code vector for a compiled function and at the end of a loop in a compiled function that checks for an interrupt (Control-C on Unix, Break/Pause on Windows).</p> <p>Note that an infinite loop that does not call functions will not be interruptable except by multiple Control-C's (Unix) and using the Franz icon on the system tray (Windows). See <i>startup.htm</i> for more information on interrupting when all else fails.</p>	<p>True unless speed is 3 and safety is 0.</p>
<p><code>internal-optimize-switch</code></p>	<p>When true, the register-linking internal optimization is performed, resulting in faster but harder to debug code.</p>	<p>True when speed is greater than 2 or debug less than 3.</p>
<p><code>optimize-fslot-value-switch</code></p>	<p>When true, calls to <b>fslot-value-typed</b> will be open-coded if possible. See <i>ftype.htm</i></p>	<p>True when speed is greater than safety.</p>
<p><code>peephole-optimize-switch</code></p>	<p>When true, the compiler performs peephole optimization. Peephole optimizations include removing redundant instructions, such as a jump to the immediately following location.</p>	<p>True when speed is greater than 0.</p>

<p>save-arglist-switch</p>	<p>If true, argument lists for compiled functions and macros will be stored (and available to tools like <b>arglist</b>).</p>	<p>True when debug is greater than 0.</p>
<p>save-local-names-switch</p>	<p>If true, the names of local variables will be saved in compiled code. This makes debugging easier.</p>	<p>True when debug is greater than 1.</p>
<p>save-local-scopes-switch</p>	<p>If true, information about when local variables are alive is saved in compiled code, making debugging more easy.</p>	<p>True only when debug is 3.</p>
<p>tail-call-self-merge-switch</p>	<p>If true, compiler will perform self-tail-merging (for functions which call themselves). See <a href="#">Section 7.6 Tail merging discussion</a> below for more information on tail-merging.</p>	<p>True if speed is greater than 0.</p>
<p>tail-call-non-self-merge-switch</p>	<p>If true, compiler will perform non-self-tail-merging (for functions in the tail position different than the function being executed). See <a href="#">Section 7.6 Tail merging discussion</a> below for more information on tail-merging.</p>	<p>True if speed is greater than 1 and debug less than 2. (This is more restrictive than tail-call-self-merge-switch described just above in this table because all references to the caller function are off the stack in this case but at least one call remains on the stack in the self-merge case.)</p>

<code>trust-declarations-switch</code>	<p>If true, the compiler will trust declarations in code (other than dynamic-extent declarations -- see next entry) and produce code (when it can) that is optimized given the declarations. These declarations typically specify the type of values of variables. If <code>nil</code>, declarations will be ignored except <code>(declare notinline)</code> and <code>(declare special)</code> which are always complied with.</p>	<p>True if speed is greater than safety.</p>
<code>trust-dynamic-extent-declarations-switch</code>	<p>If true, the compiler will trust dynamic-extent declarations in code and produce code (when it can) that is optimized given the declarations. This switch is separated from the general <code>trust-declarations-switch</code> so that dynamic-extent declarations can be ignored when in multiprocessing cases where they can be unsafe.</p>	<p>True if speed is greater than or equal to safety.</p>
<code>trust-table-case-argument-switch</code>	<p>If true, the compiler will compile suitable <b>case</b> statements in a table-driven fashion, which is faster but less safe. See the variable description (and its links) for details.</p>	<p>True if speed is 3 and safety is 0.</p>

<p><code>verify-argument-count-switch</code></p>	<p>If true, the compiler will add code that checks that the correct number of arguments are passed to a function.</p>	<p>True if speed is less than 3 or safety is greater than 0.</p>
<p><code>verify-car-cdr-switch</code></p>	<p>If true, code calling <b>car</b> and <b>cdr</b> will check that the argument is appropriate (i. e. a list). The switch is only effective on platforms which have <code>:verify-car-cdr</code> on the <code>*features*</code> list. Platforms lacking that feature ignore this switch since the verification is done differently but always.</p>	<p>True if speed is less than 3 or safety is greater than 1.</p>
<p><code>verify-funcalls-switch</code></p>	<p>If false, the compiler will compile certain calls to <b>funcall</b> in such a way that the call immediately jumps to the <b>funcall</b>'ed function's start address. This speeds up <b>funcall</b>'ing at the cost of harder debugging (the stepper and the runtime analyzer call-counter will not see such calls).</p>	<p>True (i.e. slower <b>funcall</b>'s) if speed is less than 3 or if safety is greater than 1 or if debug is greater than 0.</p>
<p><code>verify-non-generic-switch</code></p>	<p>If true, code is generated that an object of undeclared type is of the correct type when it appears as the argument to specialized functions like <code>svref</code> and <code>rplaca</code>. Thus the argument to <code>svref</code> must be a simple vector and if this switch is true, code will check that (and give a meaningful</p>	<p>True for speed less than 3 or safety greater than 1.</p>

	<p>error message). See <a href="#">Section 7.4 Argument type for a specialized function example</a> for an example.</p> <p>Note that generic has nothing to do with (CLOS) generic functions. The name way predates CLOS.</p>	
<pre>verify-symbol-value-is-bound-switch</pre>	<p>If true, code will be added to ensure that a symbol is bound before the value of that symbol is used. The result (as shown in <a href="#">Section 7.5 Bound symbol example</a> below) is that an error with an appropriate message will be signaled in code compiled with this switch true. In code compiled with this switch <code>nil</code>, behavior is undefined but it may be that no error is signaled.</p>	<p>True if speed is less than 3 or safety is greater than 1.</p>

## 7.2 Declared fixnums example

The following code illustrates the effect of this switch. We define a function that simply adds its two arguments and contains declarations that both the arguments are fixnums. When compiled with this switch returning `nil`, the function correctly returns the bignum resulting from the arguments in the example. When compiled with this switch returning `t`, the function returns a wrong answer.

```
USER(28): (defun frf (n m) (declare (fixnum n) (fixnum m))
(+ n m))
FRF
USER(29): (setq bn (- most-positive-fixnum 20))
536870891 ;; 20 less than most-positive-fixnum
USER(30): (frf bn 50000)
536920891 ;; This value is a bignum
USER(31): (proclaim '(optimize (safety 0) (speed 3)))
T
```

```

USER(32): (compile 'frf)
FRF
NIL
NIL
USER(33): (frf bn 50000)
-268385477 ;; wrong answer!
USER(34):

```

---

## 7.3 Argument count checking example

Very serious errors can occur when this switch returns `nil` and the wrong number of arguments are passed. These errors are not necessarily repeatable. We give a simple example where you get a less than useful error message, but you should be aware that much more serious (possibly fatal) errors can result from code where the number of arguments are not checked. About 3 instructions (the exact number is platform dependent and ranges from 1 to 4) are added to a function call when this switch returns `t`.

```

USER(1): (defun foo (a b c) (+ a c))
FOO
USER(2): (foo 1 2)
Error: FOO got 2 args, wanted at least 3.
[condition type: PROGRAM-ERROR]
[1] USER(3): :pop
USER(4): (proclaim '(optimize (speed 3) (safety 0)))
T
USER(5): (compile 'foo)
; While compiling FOO:
Warning: variable B is never used
FOO
T
T
USER(6): (foo 1)
#<unknown object of type number 4 @ #x8666c>
USER(7):

;; Note you might also get an error, although not one that
;; mentions the number of arguments.

```

Note that no error is signaled. Note further that it is possible for a fatal garbage collection error to result from passing the wrong number of arguments.

## 7.4 Argument type for a specialized function example

The following examples show what happens when the switch returns `t` and when it returns `nil`.

```

USER(39): (defun foo (vec n) (svref vec n))
FOO
USER(40): (setq v (list 'a 'b 'c 'd 'e))
(A B C D E)
USER(41): (foo v 3)
Error: Illegal vector object passed to svref: (A B C D E)
[1] USER(42): :pop
USER(43): (proclaim '(optimize (speed 3) (safety 0)))
T
USER(44): (compile 'foo)
FOO
NIL
NIL
USER(45): (foo v 3)
Error: Received signal number 10 (Bus error)
[condition type: SIMPLE-ERROR]

;; Or it might seem to work but return a bogus value.
[1] USER(46):

```

## 7.5 Bound symbol example

The following example shows what happens if this switch returns `nil`. In that case, the symbol-value location is simply read. If the symbol is in fact unbound, an apparently valid but in fact bogus value may be obtained. In the example, that bogus value is passed to the function `+` and an error is signaled because it is not a valid argument to `+`. However, it may be that no error will be signaled and computation will continue, resulting in later confusing or uninterpretable errors or in invalid results.

```

USER(60): (defun foo (n) (+ n bar))
FOO
USER(61): (foo 1)
Error: Attempt to take the value of the unbound variable `BAR'.
[condition type: UNBOUND-VARIABLE]
[1] USER(62): :pop

```

```

USER(63): (proclaim '(optimize (speed 3) (safety 0)))
T
USER(64): (compile 'foo)
; While compiling FOO:
Warning: Symbol BAR declared special
FOO
T
T
USER(65): (foo 1)
Error: (NIL) is an illegal argument to +
[condition type: TYPE-ERROR]

;; You may see a different error.
[1] USER(66):

```

---

## 7.6 Tail merging discussion

Consider the following function definition:

```

(defun foo (lis)
  (pprint lis)
  (list-length lis))

```

When you call **foo** with a list as an argument, the list is pretty printed and its length is returned. But note that by the time **list-length** is called, no more information about **foo** is needed but, in the interpreter at least, the call to **foo** remains on the stack. The compiler can tail merge **foo** in such a way that the call to **list-length** is changed to a jump. In that case, when **list-length** is reached the stack looks as if **list-length** was called directly and **foo** was never called at all. The side effects of calling **foo**, in this case, pretty printing the list passed as an argument, have all occurred by the time **list-length** is called.

Unwrapping the stack in this fashion is a benefit because it saves stack space and can (under the correct circumstances) avoid creating some stack frames all together. However, it can make debugging harder (because the stack backtrace printed by **:zoom** will not reflect the actual sequence of function calls, but see the discussion of ghost frames in *debugging.htm*) and it can skew profiling data (because, looking at our example, a sample taken after **list-length** is called will not charge time or space to **foo** because **foo** is off the stack).

The two switches `tail-call-self-merge-switch` and `tail-call-non-self-merge-switch` control tail merging.

`tail-call-self-merge-switch` will have an effect when its value is true at the beginning of the function being self-called (and no other time). `tail-call-non-self-merge-switch` has effect only when true at the point of the call.

---

## 7.7 Changing compiler optimization switch settings

The user may change the code which determines how any of these variables behaves by redefining the function which is the value of the variable. That function must take as arguments *safety*, *space*, *speed*, and *debug*. It is important that the function be compiled since it may be called many times during a compilation and an interpreted function will slow down the compiler. Here is the general form which modifies a switch variable. *var* identifies the switch variable you wish to change.

```
(setq var
  (compile nil '(lambda (safety space speed debug)
    form-1
    ...
    form-n)))
```

where *form-n* returns `nil` or `t` as a function of *safety*, *space*, *speed*, and *debug*.

Note that we wrapped the function definition with **compile**. Note too that such a form will not affect the compilation of the file in which it is itself compiled (since the variable will not be redefined in time to affect the **compile-file**).

For example, if the following code is evaluated at the top-level or in an initialization file, the compiler will not save local scopes if *speed* is greater than 1 or if *debug* is less than 2 or if *space* is greater than 1. The value of *safety* has no effect on the switch.

```
(setq compiler:save-local-scopes-switch
  (compile nil '(lambda (safety space speed debug)
    (declare (ignore debug))
    (cond ((> speed 1) nil)
          (< debug 2) nil)
          (> space 1) nil)
          (t t))))
```

The initial values of *safety* and *speed* are both set during image build (with **build-lisp-image**) using the *opt-speed*, *opt-safety*, *opt-space*, and *opt-debug* arguments. See *building-images.htm*.

---

## 7.8 Switch can be `t` or `nil` meaning always on or always off

The value of a compiler switch can be `t` or `nil` as well as a function. `t` is interpreted as a function that always return true and so causes the switch to always be on. `nil` is interpreted as a function that always returns false and so causes the switch to always be off. These settings are particularly useful when binding the variables during a specific compilation.

---

## 8.0 Pointers for choosing speed and safety values

What values should you choose for speed and safety? It is tempting to set speed to 3 and safety to 0 so that compiled code will run as fast as possible, and devil take the hindmost. Our experience is that people who do this say that Allegro CL is fast but lacks robustness, while people who use the more conservative default settings of 1 and 1 feel that Allegro CL is very robust but occasionally seems a bit sluggish. Which you prefer is for you to decide. The following points should be considered.

1. If you are going to set speed globally to 3, we *strongly* discourage you from also globally setting safety to 0 rather than 1. There are only 3 differences between the two safety settings: at safety 0, (1) argument count checking is disabled; (2) interrupt checking is disabled; and (3) sums and differences of fixnums are assumed to be fixnums. Thus you may not be able cleanly to break out of an infinite loop; passing the wrong number of arguments may cause unrepeatable, possibly fatal errors; and adding fixnums whose sum is a bignum will silently produce the wrong answer. We recommend that those switches should only be set in an unsafe manner when compiling functions where it is very unlikely that the code is incorrect. You can set safety to 0 when compiling such functions by use of a declaration within the `defun` form.
  2. If strange errors occur whose cause cannot be discovered, recompile the code at a more safe setting of speed and safety and run it again. An error will usually occur (not necessarily in the same place), the error message should be more informative, and the debugger should have more information. If no error occurs, it may be that you have an incorrect declaration (e.g. declaring a value to be a single-float when it is in fact something else, like `nil`). Look particularly at initial values, making sure the initial value is of the declared type.
  3. You might consider resetting the switch variables to `t` or `nil` (as appropriate) as that can ensure the switch has a known, unambiguous value unaffected by declarations within a function definition or anything else.
- 

## 8.1 Optimizing for fast floating-point operations

The compiler in Allegro CL has the ability to compile floating-point operations in-line. In order to take full advantage of this feature, the compiler must know what can be done in-line. For it to know this, the user must, through declarations, inform the compiler of the type of arguments to operations. The compiler will attempt to propagate this type information to other operations, but this is not as easy as it might seem on first glance. Because mathematical operations in Common Lisp are generic, that is they accept arguments of any type - fixed, real, complex - and produce results of the appropriate type, the compiler cannot assume results in cases when the user may think the situation is clear. For example, the user may know that only positive numbers are being passed to `sqrt`, but unless the compiler knows it too, it will not assume the result is real and not complex. The compiler can tell the user what it does know and what it is doing. See [Section 9.0 Help with declarations: the `:explain` declaration](#) below. With this information, the user can add declarations as needed to speed up the compiled code. The process should be viewed as interactive, with the user tuning code in response to what the compiler says it knows.

---

## 8.2 Supported operations

The compiler will expand in-line (open-code) a floating-point numeric function only if the function is one which it knows how to open-code, and the types of the function's arguments and result are known at compile time to be those for which the open-coder is appropriate. Finally, at the point of compilation the compiler switch function which is the value of `comp:trust-declarations-switch` must return true. The default version of this function returns true if `speed` is greater than `safety` but users can change the values for which this switch returns true. **`explain-compiler-settings`** can be called to see what the value of that switch (and all others) will be given the current values of `safety`, `space`, `speed`, and `debug`.

The floating-point functions below are subject to opencoding. In each case, the function result must be declared or derivable to be either single-float or double-float. (Note that in this implementation, the type `short-float` is equivalent to single-float and `long-float` is equivalent to double-float.) The arguments to the arithmetic and trigonometric functions must be specifically one or the other of the two floating types, or signed integer constants representable in 32 bits, or else computed integer values of `fixnum` range. If these conditions are not met, the function will not be open-coded and instead a normal call to the generic version of the function will be compiled. Note that it is not sufficient to declare a value to be a float, it must be declared as either a single-float or a double-float.

We are often asked exactly which functions will opencode in Allegro CL. Unfortunately, it is not easy to provide an answer. Firstly, because the compiler is different on each different architecture, the answer is different in different implementations. Secondly, the same function may opencode in one case but not in another on the same machine because of the way it is affected by surrounding code. However, the following functions are candidates to be open-coded in most platforms or in the platforms noted.

- The four binary arithmetic functions: `+` `-` `*` `/`

- The four unary arithmetic functions: + - \* /
- The simple trigonometric functions: **sin**, **cos**, **tan** (Windows on Intel x86/Pentium or equivalent only)
- One-argument **float** of a fixnum.
- **abs** (all platforms) and **sqrt** (HP, Intel x86/Pentium or equivalent, Sparc),
- **aref** and **setf** of **aref** of (simple-array single-float (\*))
- **aref** and **setf** of **aref** of (simple-array double-float (\*))

Note that the list is not exhaustive in either direction. Not all the listed functions and operations opencode on all machines and functions and operations not listed do opencode on some machines. The `:explain` declaration described in the next section assists with determining what did and did not opencode.

---



---

## 9.0 Help with declarations: the `:explain` declaration

The compiler must be supplied with type declarations if it is to open-code certain common operations without type-checking. The compiler includes a type propagator that tries to derive the results of a function call (or special operator) from what it knows about its arguments (or subforms). The type propagator greatly reduces the amount of type declaration necessary to achieve opencoding. However, the programmer may need to examine the inferences made by the propagator to determine what additional declarations would increase code speed. Although supplying type declarations to the compiler is very simple, it is a task surprisingly prone to error. The usual error is that well-intentioned declarations are insufficient to tell the compiler everything it needs to know.

For example, the programmer might neglect to declare the seemingly obvious fact that the result of a certain `sqrt` of a double-float is also a double-float. However, `sqrt` in Common Lisp returns a complex if its argument is negative, so the compiler cannot assume a real result. The only impact of insufficient declarations is that some open-coders will not be invoked. However, it can be awkward for the user to determine whether or not any particular call was open-coded, and if not, why. `trace` and `disassemble` will provide the information, but these are clumsy tools for the purpose.

In order to provide better information about what the compiler is doing, a new declaration, `:explain`, has been added to Allegro CL.

---

### `:explain`

## Declaration

**Arguments:** `[:calls / (:calls t) / (:calls nil)] [:types / (:types t) / (:types nil)] [:boxing / (:boxing t) / (:boxing nil)] [:variables / (:variables t) / (:variables nil)] [:tailmerging / (:tailmerging t) / (:tailmerging nil)] [:inlining / (:inlining t) / (:inlining nil)]`

This declaration instructs the compiler to report or not to report information about argument types and non-in-line calls, boxed floats and variables stored in registers, tail-merging, and why inlining might not have succeeded. Reporting is enabled when a quality appears alone or in a list with `t`. It is disabled when a quality appears in a list with `nil`. Initially, no `:explain` qualities are enabled.

This declaration may be placed anywhere that a normal declaration may be, and can be proclaimed (with **proclaim** or **declaim**). The compiler will report information within the scope of the declaration. Note that results may differ from one platform to another. Again, the general form of the declaration is (these forms must be placed in a location where declares are allowed, of course):

```
(declare (:explain :quality ...)) ;; explanation for :quality
                                   ;; will be output

;; or
(declare (:explain (:quality t) ...)) ;; explanation for :quality
                                       ;; will be output

;; or
(declare (:explain (:quality nil) ...)) ;; explanation for :quality
                                         ;; will not be output

;; Thus
(declare (:explain :types (:boxing nil) (:variables t) :tailmerging))
;; will enable explaining for :types, :variables, and :tailmerging
;; and disable it for :boxing. Explaining for :calls is not affected
;; (off if it was already off, on if it was already on).
```

The arguments control various kinds of reporting the compiler will make during compilation. The arguments may appear in either order, and either may be omitted. By default, no information of either type is printed. The declaration causes the compiler to print information during compilation, and obeys normal declaration scoping rules. It has no effect on code generation. Here are the various `:explain` qualities:

- `:types`: when this quality is in effect, the compiler will report for each function call it compiles (in-line or not) the types of each argument along with the result type. Further information and examples are in *Calls and types explanation* in *compiler-explanations.htm*.
- `:calls`: when this quality is in effect, the compiler will report when it generates code for any non-in-line function call. `:types` and `:calls` operate independently. Further information and

examples are in *Calls and types explanation* in *compiler-explanations.htm*. `:calls` should not be used with `:inlining` since calls information is provided by the `:inlining` explanation.

- `:boxing`: when this quality is in effect, the compiler will tell you when code is generated to box a number if it is possible for the code not to be boxed. A number is 'boxed' when it is converted from its machine representation to the Lisp representation. For floats, the machine representation is one (for singles) or two (for doubles) words. Lisp adds an extra word, which contains a pointer and a type code. For fixnums, boxing simply involves a left shift of two bits. For bignums which are in the range of machine integers, boxing again adds an additional word. Further information and examples are in *Boxing explanation* in *compiler-explanations.htm*. `:boxing` should not be used with `:inlining` since boxing information is provided by the `:inlining` explanation.
- `:variables`: when this quality is in effect, the compiler will report whether local variables (in function definitions) are being stored in registers or in memory. Since storing variables in registers results in faster code, the information printed when this variable is in effect may help in recasting function definitions to allow for more locals to be stored in registers. Further information and examples are in *Variables explanation* in *compiler-explanations.htm*.
- `:tailmerging`: when this quality is in effect, the compiler will report why a tail-merge is or is not being done for a function in tail-position. Further information and examples are in *Tail-merging explanation* in *compiler-explanations.htm*.
- `:inlining`: when this quality is in effect, information about why a function you might expect to be inlined was not in fact inlined by the compiler. Further information and examples are in *Inlining explanation* in *compiler-explanations.htm*. (Essentially all the information printed by the `:explain :calls` and `:explain :boxing` declarations is printed by the `:explain :inlining` declaration, so you should use either `:inlining` or `:boxing/:calls`, but not both.)

The reports are printed during the code generation phase of compilation. Code generation occurs fairly late during compilation, after macroexpansion and other code transformations have taken place. The function calls explained by the compiler will therefore deviate in certain ways from the original user code. For example, a two-operand `+` operation is transformed into a call to the more efficient `excl::+_2op` function. In general, the user should easily be able to relate the code generator's output to the original code.

The code generator works by doing a depth-first walk of the transformed code tree it receives from early compiler phases. In this tree walk, the `:types printout` happens during the descent into a branch of the tree. The `:calls printout` happens as the instructions are actually generated, during the ascending return from the branch.

The reason `:explain` is implemented as a declaration is so the user can gain fairly fine-grained control of its scope. This can be important when tuning large functions. The tool does require editing the source code to be compiled, but presumably the user is in the process of editing the code to add declarations anyway. These options may also be enabled and disabled globally by use of **proclaim** or **declaim**, although they may produce a lot of output.

The lines of explanation output are labeled with abbreviations which identify what type of explanation is being produced and what the compiler is doing. These abbreviations are listed in the *compiler-explanations.htm*.

---



---



---

## 10.0 Other declarations and optimizations

There are other declarations which affect (or in some cases do not affect) the operation of the compiler, as we describe under the next several headings.

---

### 10.1 Inline declarations ignored

The `inline` declaration is ignored by the compiler. At appropriate settings of speed and safety, the compiler will inline whatever it can. Only predefined system functions can be inlined. User defined functions are never compiled inline. (The compiler will observe the `notinline` declaration, however, so you can suppress inlining of specific functions if you want.)

We have been asked why the inline declaration is ignored. It is not that we believe that inlining user functions does not provide any advantages, it is that we believe that there are other improvements that will provide more advantages. Because we have now implemented compiler environments (see *environments.htm*), we are in a better position to implement inlining of user functions in a later release.

Note that inlining is not an unmixed blessing. It increases the amount of space used by a function (since both the function definition and the block to stick in when the function is inlined have to be created and stored) and it makes debugging harder (by making the compiled code diverge from the source code). It is also prone to subtle errors (on the programmers side) and bugs (on our side).

---

### 10.2 Defstruct accessors

Defstruct accessors will be compiled inline under two conditions:

1. The structure is *not* of type list. (Accessors for structures of type list are never inlined).
2. The values of speed and safety are such that the compiler switch `verify-non-generic-`

`switch` returns `nil`. Using the default value, that switch will return `nil` when `speed` is 3 and `safety` is 0 or 1.

---

## 10.3 Stack consing, avoiding consing using `apply`, and stack allocation

### Stack consing

On some machines, stack consing of `&rest` arguments, closures, and some lists and vectors is supported if declared as `dynamic-extent`. (Stack consing means that objects are stored on the stack and not written to memory. This can provide a significant space saving if the objects are truly temporary, which they often are.) Here is an example of such a declaration for an `&rest` argument. The function `foo` checks whether the first argument is identical to any later argument.

```
(defun foo (a &rest b)
  (declare (dynamic-extent b))
  (dolist (val b)
    (if* (eq a val) then (return t))))
```

Please note that care should be exercised when using stack consing and multiprocessing, since a switch between one thread and another causes (part of) the stack to be saved, so the stack-consed objects and values are not available to the thread being switched into. For a binding or object that has dynamic extent, that extent is only valid when Lisp is executing on the thread that creates the binding or stack-conses the object. When (on Windows) a thread switch is executed, the extents of all dynamic-extent data and bindings are temporarily exited, to be re-established when another thread switch returns to the original thread. Since on Windows separate Lisp lightweight processes run on separate threads, it is important that dynamic-extent data (which may be stack-consed) not be referenced while executing on a different thread. On Unix, stack-groups were used rather than threads, and so a **process-wait** *wait-function* argument should never be declared dynamic-extent, since it was funcall'ed from other stack-groups. However, on Windows, wait functions are run only in their own threads, so stack consing in wait functions should work.

Dynamic-extent declarations are only observed at values of `safety`, `space`, `speed`, and `debug` for which `trust-dynamic-extent-declarations-switch` returns true.

If a call to **make-list** has a constant size, declarations are trusted, the list is made the value of a variable, and the variable is declared as dynamic-extent, then it will be stack-allocated and initialized. The *initial-value* keyword can be used to specify the value. An attempt to make a list of a variable size with **make-list** will result in heap consing.

Dynamic-extent argument properties are automatically declared on all **defun** forms. This gives the compiler the ability to make assumptions about the dynamic extent use of arguments passed into these functions, and to generate more efficient code. The compiler has always tracked these properties for functions that it knows about, e.g. **mapcar**, and this new facility extends the interface to user-defined functions and to redefinitions. Warnings are also provided for redeclared definitions and for definitions that occur after the function's first usage. To allow declaration before the first use, a new macro called **defun-proto** is provided.

### Avoiding consing with **apply** using a **&rest**

In certain cases **apply** is now compiled more efficiently, to remove consing. This is the so-called *applyn* optimization. Consider the following code:

```
(defun foo (x &rest y)
  (apply some-fun 1 x 2 y))
```

The **&rest** argument is used for nothing more than to supply a variable length argument list to **apply**. This case is now compiled in such a way that

1. The last argument to **apply** is not actually used, but an index to the n'th argument to **foo** is compiled instead.
2. The **&rest** argument is considered dead and as if declared ignored.
3. All aspects of the function are preserved (e.g. possible argument checking for a minimum but not a maximum number of arguments, etc.)

In this optimized case, the code works exactly as it did when the **&rest** argument was consed, but without the consing. Circumstances that will cause this optimization to not be used are if the **&rest** argument:

- is declared special,
- is a closed-over variable,
- is set.
- is used anywhere except as the last argument to **apply**.

**Optimization hint:** If you have a function like

```
(defun wrap-it (flag &rest x &key &allow-other-keys)
  (when flag
    (setq x (list* :new-key 10 x)))
  (apply 'wrapped flag x))
```

then the optimization will not take effect. If non-consed operation is desired, then the following modification will allow the optimization:

```
(defun wrap-it (flag &rest x &key &allow-other-keys)
  (if flag
    (apply 'wrapped flag :new-key 10 x)
    (apply 'wrapped flag x)))
```

## Stack allocation

The following types of vectors can now be stack-allocated:

element type	initializable? (see below)
t	yes
(unsigned-byte 32)	yes
(signed-byte 32)	yes
(unsigned-byte 16)	no
(signed-byte 16)	no
(unsigned-byte 8)	no
(signed-byte 8)	no
character	no
(unsigned-byte 4)	no
bit	no
excl::foreign	yes
single-float	no

To get a stack-allocated vector, the following coding practice should be used:

```
(declare (optimize <values that make trust-declarations-switch
```

```

true>))
  ;; with initial variable value, (speed 3) (safety 1) will work
...
(let ((x (make-array n <options>)))
  (declare (dynamic-extent x)) ...)

```

where *n* is a constant integer, and *options* are limited to `:element-type` (and `:initial-element`, if the array is **initializable** according to the table above). All other forms might cause heap allocation of the array.

Other functions that allow stack-consing if conditions are right are **cons**, **list**, **list\***, **make-list**, **with-stack-fobject**, **with-stack-fobjects**, **with-stack-list**, and **with-stack-list\***.

## 10.4 Adding typep-transformers

A typep-transformer allows the compiler to transform a form like

```
(typep x 'foo)
```

into a form like

```
(funcall some-function x)
```

For example, the Allegro CL compiler will already transform typep forms where the type is defined as:

```
(deftype foo () `(satisfies foop))
```

into

```
(funcall 'foop x)
```

The ability to add typep-transformers described here allows types defined with a more complicated syntax than `(satisfies 'some-function)` to be similarly transformed. The user must supply the appropriate predicate function and call the following function to associate the type with the predicate.

The function **add-typep-transformer** takes a type and a predicate which will allow the compiler to transform forms like

```
(typep x 'type)
```

into the form:

```
(funcall <predicate> x)
```

The function **remove-typep-transformer** removes the association between the type and the predicate.

For example, suppose we have defined a type called `angle`, which is a normalized angular measurement:

```
(deftype angle () '(real #.(* -2 pi) #.(* 2 pi)))
```

Suppose further that we have a time critical function that takes two arguments, checks to be sure they are angles, adds them together, checks to make sure the result is an angle, and returns it:

```
(defun add-two-angles (a b)
  (declare (optimize (speed 3)))
  (unless (typep a 'angle)
    (error "~s is not an angle" a))
  (unless (typep b 'angle)
    (error "~s is not an angle" b))
  (let ((sum (+ (the angle a) (the angle b))))
    (unless (typep sum 'angle)
      (error "Sum (~s) of angles is not an angle" sum))
    sum))
```

As is, 10,000 calls of this function (given legal arguments) takes about 1.5 CPU seconds (on my test machine). Suppose we're unhappy with that and want to speed it up by adding a `typep-transformer`, without having to change the coding of **add-two-angles**. Further suppose that we're usually dealing with single precision floating point numbers. Here's a way we can do it. We first define our predicate function. Note that we put the most likely case (`single-float`) as the first choice in the **typecase** form.

```
(defun anglep (x)
  (declare (optimize (speed 3) (safety 0)))
  (typecase x
    (single-float (and (>= (the single-float x)
```

```
#.(float (* -2 pi) 0.0s0))
```

```
(<= (the single-float x)
```

```
#.(float (* 2 pi) 0.0s0))))))
```

```
(fixnum (and (>= (the fixnum x) #.(truncate (* -2 pi)))
(<= (the fixnum x) #.(truncate (* 2 pi))))
(double-float (and (>= (the double-float x)
#.(float (* -2 pi) 0.0d0))
(<= (the double-float x)
#.(float (* 2 pi) 0.0d0))))))
```

We then call **add-typep-transformer** to make the compiler aware of our predicate:

```
(excl:add-typep-transformer 'angle 'anglep)
```

Now if we recompile **add-two-angles** and call it another 10,000 times with the same arguments, it only takes about .25 CPU seconds, a 6 fold improvement (you may see a different speedup ratio).

## 10.5 Compiler handling of top-level forms

See the discussion in *implementation.htm* and the description of `*cltl1-compile-file-toplevel-compatibility-p*` for information on special handling of certain top-level forms in a file. The issue is whether the forms are treated (during the compile) as if wrapped in an

```
(eval-when (compile) ...)
```

A top-level form in a file is one which is not a subform of anything except perhaps **progn**. In CLtL-1 CL, top-level forms involving calls to the following functions were treated as if wrapped in such an **eval-when** when compiled. In ANSI CL, they are not. You can arrange to have the CLtL-1 behavior as described in *implementation.htm*. The affected functions are:

```
make-package
  shadow
  shadowing-import
  export
  unexport
  require
  use-package
  unuse-package
  import
```