Corman Lisp

Common Lisp Development Environment

Version 3.1

User Guide

for Windows 32-Bit Operating Systems

December 30, 2018

Contents

Corman Lisp User Guide

1 Introduction

This document is intended for use with Corman Lisp, a Common Lisp development environment for Microsoft Windows operating systems running on Intel platforms. Corman Lisp consists of a Common Lisp native code compiler for Intel processors, 80x86 assembler and disassemble, incremental linker and multi-window text editor. It requires a system running a Microsoft Windows operating system (such as Windows ME, Windows XP, or Windows 2000). It is fully integrated with the Win32 API, and all the Windows API functions are readily available from Lisp.

Corman Lisp incorporates state-of-the-art compiler technology to bring you a Common Lisp system unmatched on Windows platforms. Among the highlights of Corman Lisp:

- **Fast multi-generational garbage collector**. A state-of-the-art garbage collector supports extremely fast collection of short-lived objects, avoiding pauses for collection (which normally takes only a few milliseconds). The collector also supports compaction (automatic defragmentation of the heap), weak pointers, finalization functions and per-thread allocators.

- **No interpreter.** Code is compiled 100% of the time. It feels like an interpreter, and acts like one, but always gives you the best possible performance of compiled, optimized code.

- **Compilation is extremely fast.** The sources to Corman Lisp (consisting of 90 common Lisp source files and 40,000 lines of code) load, compile and save to disk in under 10 seconds on a 1 GHz workstation. You can try it yourself if you like (all sources are included).

- **Foreign Function Interface.** All functions in DLLs may be called directly from Corman Lisp code, including all Win32 API functions. Most Win32 data types and many Win32 API functions come predefined and ready to be used in the WIN32 package. A special C declaration parser is built-in which allows Win32 C API functions to be used with little or no modifications (obtained from Microsoft C header files).

- **Multi-threading.** The compiler has been designed to support multiple threads of execution, with special variable bindings on a per thread basis. These threads are managed by the operating system, affording you the full advantage of the features provided by the operating system. Multiple CPUs are supported, with lisp threads running simultaneously on multiple processors.

- **Foreign Callbacks into Lisp.** The Foreign Function Interface supports definition of callback functions which can be used as Windows procedures or for other uses. This allows straight Win32 SDK applications to be built, exploiting the whole power of the operating system. These callbacks can be interactively edited and replaced, even while being used by the operating system or another program. Several sample programs are included which were translated line by line from C language programs in Charles Petzold's *Programming Windows 95*. These aren't intended to demonstrate good Lisp programming style—rather to show it can be done.

- **CLOS and Metaobject Protocol.** The Corman Lisp CLOS implementation is adapted from Art of the Metaobject Protocol, and allows you to take advantage of object-oriented techniques where appropriate. It has a MOP (Metaobject Protocol) of great flexibility, and is very well documented in the book *Art of the Metaobject Protocol* by Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. This implementation has been significantly rewritten, enhanced and optimized, with native compiler support for CLOS instances and method compilation.

- **Optimizing Compiler.** Several key optimizations allow the compiler to generate very fast code. These include automatic inlining of small commonly used Common Lisp functions, automatic stack allocation of lexical variables which do not persist after the function exits, and tail recursion elimination.

- **Integrated Intel Assembler.** Most Intel assembler instructions are supported, and can be used to do special-purpose tasks from Corman Lisp for which a Lisp function is not possible.

- **Source Code.** The source code for Corman Lisp is included with this release. Most of this source is in Common Lisp, and self-compiles. The bootstrap kernel code is written in C/C++, and Intel assembler, and is included with the standard Corman Lisp release. Complete instructions for rebuilding Corman Lisp from sources are included with this package.

- **COM Server.** Corman Lisp is built as an in-process COM server, and can be accessed as such to increase its flexibility for use in different programming environments. The included development environment interacts with Corman Lisp exclusively via COM interfaces. The COM interface information is included in the Corman Lisp sources in this package.

- **Integrated Development Environment (IDE).** A multi-windowed text editor for editing, compiling and executing Corman Lisp programs is included. The IDE runs in a different thread than Corman Lisp programs, always allowing full control even while Corman Lisp is running. Multiple-level Undo/Redo is now supported.

- **Common Lisp Hyperspec Support.** The ANSI Common Lisp standard, along with much useful accompanying documentation, has been made available in a package called the Common Lisp Hyperspec. The is courtesy of Kent Pitman and Xanalys. You may directly browse the entire Hyperspec from within Corman Lisp, and in addition, all Corman Lisp symbols are linked to the Hyperspec pages that define them.

- **SAVE-IMAGE function.** The SAVE-IMAGE function (as well as its LOAD-IMAGE counterpart) allow saving the entire Corman Lisp heap to a file, for later reloading. This capability may be used to build standalone applications which do not require the presence of the IDE.

- **SAVE-APPLICATION function.** The SAVE-APPLICATION function allows you to build standalone applications with Corman Lisp which consist of a single .EXE file or (optionally) an .EXE file and a .DLL file. These can be regular Windows applications, or console applications.

- **COMPILE-DLL function.** The COMPILE-DLL function allows you to build standard Windows DLLs (dynamic link libraries) with Corman Lisp. These may be easily linked to other (non-lisp) applications, or configured for use as plug-ins with applications which supports plug-ins.

- **Source Code Management.** The compiler remembers the file and line number of every Lisp function you load. You can use FIND-SOURCE to automatically bring up the source code for editing.

2 Licensing and Copyright Information

All files provided with the release package of Corman Lisp are

Copyright (c) Corman Technologies Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

(With the exception of several public source files which are clearly marked as such at the top of each file.)

http://www.cormanlisp.com/

In January 2015, Roger Corman agreed to release the Corman Lisp sources under the MIT license. Roger provided the source code for Corman Lisp to Zach, who updated the license information and posted the code to github:

https://github.com/sharplispers/cormanlisp

3 How To Reach Us

Any comments or questions may be directed via e-mail to:

mailto:admin@cormanlisp.com

The Corman Lisp web site is at:

http://www.cormanlisp.com/

The Corman Lisp source code is at:

https://github.com/sharplispers/cormanlisp

4 Installing Corman Lisp

1. Double-click on the installation file.

2. The setup program will run, and guide you through the installation process.

The installer creates the **%CORMANLISP_HOME%** environmental variable which contains the path to the installation directory. It can be used for integrating Corman Lisp into your project.

5 Quick Start

This section is intended to briefly lead you through writing and running a small Common Lisp program with Corman Lisp.

Before starting, look at your keyboard. You probably have two keys labeled **Enter**. One of these is typically to the right of the character keys, and the other is on the numeric keypad. The Corman Lisp IDE distinguishes between these two keys. We will refer to the one near the character keys as the **Enter** key, and the one on the numeric keypad as the **Numeric Enter** key.

In the Corman Lisp IDE, you can execute commands by pressing either the **Numeric Enter** key, or by pressing **<Shift>-Enter**. Use whichever works best for you. The non-shifted **Enter** key is used in the normal way for text editing i.e. inserting a line break. In the Corman Lisp IDE, when you press **Enter**, the system may automatically indent 4 spaces for each open left parenthesis. We will refer to the **Numeric Enter** key, or the **<Shift>-Enter** key, as the **Execute** key.

1. **Start Corman Lisp by selecting it from the Start menu.**

   If Corman Lisp has started correctly, a Worksheet window should appear, and the status bar at the bottom of the main frame should say Ready.

2. **Enter a Corman Lisp expression.**

   Try typing:

   ```
   (+ 20 30)
   ```

   This command invokes the Common Lisp + function which returns the sum of the passed arguments. After you have typed the expression, while your text cursor is still positioned to the right of the closing parenthesis, note that the opening and closing parentheses are highlighted. This shows which level of parentheses are matching, and also designates a Lisp selection. To execute the selected expression, press the **Execute** key.

   All Corman Lisp output is in blue text. What you type is shows up as normal (black) text.

3. **Create a Common Lisp function.**

    Type:

    ```
    (defun factorial (x)
        "Returns the factorial of N, where N >= 1"
        (if (= x 1)
            1
            (* x (factorial (- x 1))))))
    ```

    While typing this function, use the **Enter** key to end each line.

    After the whole function has been entered, leave the text cursor positioned just after the last closing parenthesis. Press **Execute**.

    The function FACTORIAL has now been defined.

4. **Execute the function.**

    Type:

    ```
    (factorial 100)
    ```

    The factorial of 100 is returned (this is a large number). You can try it with other numbers as well. Try (factorial 1000) if you are brave. If you try too big a number, and the system appears to hang (the editor will still be active, but it will not evaluate anything you try to execute) try **<Control>-Break**. This key combination may always be used to break out of a long computation.

5. **Get function documentation.**

    Type:

    ```
    (documentation 'factorial 'function)
    ```

    The system will display documentation that you defined for the function.

6. **Time the function.**

    Execute:

    ```
    (time (factorial 100))
    ```

    The system will execute the expression and return the value, as well as print some information regarding execution time.

7. **Trace the function.**

   Execute:

   ```
   (trace factorial)
   (factorial 10)
   ```

   The system will display information about arguments are returned values for each invocation of function FACTORIAL.

8. **Save the function you have defined.**

   Select the **New** command from the **File** menu. Name the new file **factorial.lisp**.

   Select **Corman Lisp Worksheet** from the **Window** menu to return to the worksheet (or just click on its window). Select the function definition (from step 3) by highlighting the whole thing. An easy way to do this is to double-click just to the right of the last closing parenthesis or to the left of the first open parenthesis of the function definition. Double-clicking here will cause the entire Lisp expression to be selected.

   Execute the **Copy** command via the **Edit** menu or pressing **<Control>-C**.

   Select the file **factorial.lisp** from the **Window** menu or by clicking on its window.

   Execute the **Paste** command via the **Edit** menu or pressing **<Control>-V**.

   The function definition should be displayed in the **factorial.lisp** window.

   Select **Save** from the **File** menu to save the file.

9. **Load the new file.**

   Choose **Execute File…** from the **File** menu, and select your file **factorial.lisp**.

   This loads the function you defined. (It was already loaded, but this is for demonstrative purposes.) You can accomplish the same thing by executing the following expression:

   ```
   (load "factorial.lisp")
   ```

10. **Close the new file.**

   You can use **Close** from the **File** menu, or click the window's close box.

11. **Find the function for editing.**

If you wish to edit any loaded function, Corman Lisp includes a way to automatically bring up the function for editing.

Execute:

```
(db:find-source 'factorial)
```

This should open the file `factorial.lisp` in an edit window, and places the text cursor at the beginning of the `FACTORIAL` function.

Important Notes

- You may have any number of files open at once.

- There is no difference between the **Lisp Worksheet** and any other file. Every open file may act as a worksheet, in that Lisp expressions may be executed from them. Lisp output will, however, always be directed to the file called **Lisp Worksheet**. This allows you to easily select and execute portions of your files while you are editing them without risk that Lisp output will accidentally get saved with the file.

6 Files in This Release

When you install Corman Lisp, you selected an installation directory. We refer to this as the Corman Lisp directory. There are subdirectories located within this directory, in which many of the Corman Lisp files are stored.

Files located in the Corman Lisp directory:

CormanLisp.exe

This is the Corman Lisp Integrated Development Environment (IDE) application. Double click on it to launch Corman Lisp.

CormanLispServer.dll

The Corman Lisp kernel. It should be located in the Corman Lisp installation directory. It is a COM server for the Corman Lisp system, and will be registered as such by your system the first time you run Corman Lisp.

CormanLisp.img

This contains the remainder of the Corman Lisp system. It is a Lisp image file, produced by loading the Corman Lisp system and using SAVE-IMAGE.

CLConsole.exe

This is a simple console application which allows you to interact with Corman Lisp via a console interface. We expect you will find the IDE much nicer to use than this. However, this is an example of how the Corman Lisp COM server may be used to interact with Corman Lisp from another application. Source (in C++) is provided in the clconsole directory. In addition to being used by itself to run a lisp console, this file also serves as a template application to generate console-based applications using SAVE-APPLICATION.

CLConsoleApp.exe

This file is similar to CLConsole.exe, except that it contains the Corman Lisp kernel (**CormanLispServer.dll**) linked as a static library. This file serves as a template application to generate applications using SAVE-APPLICATION.

CLBoot.exe

This is a simple Windows application which has no user interface. It may be used to load and run a Lisp image, but the Lisp program must handle all the input and

output by itself. This file serves as a template application to generate applications using SAVE-APPLICATION.

CLBootApp.exe

This file is similar to CLBoot.exe, except that it contains the Corman Lisp kernel (CormanLispServer.dll) linked as a static library. This file serves as a template application to generate applications using SAVE-APPLICATION, with the standalone option set to true.

Init.lisp

This file always gets executed when the lisp top level loop starts. This typically is used to customize the console or IDE by executing lisp expressions at start. It should be noted that this file gets executed by the every instance of the Corman Lisp on the system. Users are encouraged to use an user specific *initialization file* whenever possible (see the next section for details).

On-line documentation can be found in the subdirectory called **documentation**.

User Specific Files and Directories

Starting from version 3.1 during its functioning Corman Lisp IDE creates directory which it uses for its output (e. g. Lisp image saving, creation of executable, writing crash dumps etc). We will call this dierctory **output directory** in the manual. For console version of the Corman Lisp **output directory** equals to a current working directory (by default, the directory, from which the executable was started). In previous versions of the Corman Lisp the above mentioned **installation directory** was used for this facilities. The directory is specific for every user profile on Windows. Its default location is

`%USERPROFILE%\Documents\Corman Lisp\`

or

`%USERPROFILE%\My Documents\Corman Lisp\`

on older versions of Windows. This directory gets created by the Corman Lisp IDE automatically on launch.

`Lisp Worksheet`

> This file normally gets loaded as the worksheet when you launch Corman Lisp. If you remove or delete this file, a new one will automatically be created when you restart Corman Lisp. This file is located in the above mentioned Corman Lisp **output directory**. It is used to be located in the **installation directory** in previous releases.

`%USERPROFILE%\corman-init.lisp`

> This file considered to be an **initialization file**. This file gets executed when the top-level loop starts. It is very similar by its function to the **init.lisp** in the **installation directory** but might be used for user specific customization instead of system-wide ones. Users of the Corman Lisp are encouraged to use this file instead of **init.lisp** whenever possible.

Source Code

All source code to Corman Lisp that is provided with this release, unless marked otherwise, is copyrighted by Corman Technologies. It is for non-commercial use only, and it is forbidden to reproduce it for commercial purposes. The source code may not be modified or redistributed by anybody except with written permission from the author. Source code includes all files with the extension `.lisp`, `.cpp` or `.h`.

The source code to Corman Lisp is mainly written in Common Lisp, and is provided with this release. It can easily be browsed from within Corman Lisp, either by using the **Open...** command from the **File** menu, or by executing the `DB:FIND-SOURCE` command. The Common Lisp sources for Corman Lisp are in the `SYS` directory. You should never `LOAD` any of these files, as they are loaded in a very order-dependent manner during the building of Corman Lisp. Additional sources (which are loaded optionally via `REQUIRE` or `LOAD`, can be found in the `MODULES` and `LIBRARIES` directories.

The Corman Lisp kernel is written in C/C++/x86 assembler. It is compiled with Microsoft Visual Studio 2005 (which handles all three of those languages at once). The kernel source is used to build the shared DLL named `CormanLispServer.dll` (which is an in-process COM server). The kernel code implements the following functionality:

- COM Server interface

- Bootstrap Common Lisp code (to start the engine)

- Memory management (including the ephemeral garbage collector)

- Code Generation (generating x86 machine code)

- Kernel Library functions

- GZIP compression (used by `SAVE-IMAGE`, `COMPILE-FILE`).

The majority of the kernel library functions are replaced by functions written in Common Lisp when the `CormanLisp.img` file is built.

The code generation module has hooks to call Common Lisp code as it compiles each expression and sub-expression. Eventually most code generation will probably be performed by Common Lisp code which will implement more advanced compiler optimizations.

We have no plans to move the memory management functions to Common Lisp, because we see no need for this and because the nature of the Common Lisp compiled functions is that they get moved by the garbage collector. The garbage collection code would have to be handled specially.

We expect the COM server interface to be enriched by Common Lisp code, which will be able to easily create and export COM interfaces in upcoming versions.

We have included the C/C++/x86 sources for the kernel with this release. This has been compiled with Microsoft Visual Studio .NET 2002.

The source code to the Corman Lisp IDE is not provided, and we do not plan to make it publicly available at this time. The CLConsole and CLBoot applications both include source and provide examples of using Corman Lisp from C++ applications.

Recompiling Corman Lisp from Sources

To recompile the CormanLisp system from Common Lisp sources, perform the following steps:

1. Optionally backup or rename the file **CormanLisp.img** (in the Corman Lisp root directory).

2. Execute the operator:

    ```
    (ccl:compile-cormanlisp-image)
    ```

This operator takes no arguments, and replaces the file CormanLisp.img in the root install directory.

7 Integrated Development Environment (IDE)

The Corman Lisp kernel is a COM server that does not contain a user interface. The Corman Lisp Integrated Development Environment provides an application which runs the Corman Lisp system, and integrates several features that are useful for developing Common Lisp programs.

- A multi-window text editor. There is no limitation on file size. Color coding, parentheses matching, auto-indenting and other features are used to assist you in writing Common Lisp source code. You can set font style, size and color, and tab size.

- Debugging facilities.

- Interactive function execution.

- Status bar.

- HTTP Browser (based on Internet Explorer) can be used to browse documentation files or anything else, and can be programmed by Lisp code.

- Ability to try out Common Lisp programs by running them directly from the IDE.

- IDE runs in a separate thread.

- Menu items for loading and executing Lisp programs.

The environment provides a *worksheet* approach to Common Lisp development. Rather than having a window that emulates a console (e.g. the *Listener* in Allegro Common Lisp or LispWorks for Windows), the worksheet approach does not emulate a console. Any number of text windows may be open, and any Common Lisp code in any open window may be executed at any time. The user typically enters a Common Lisp function or expression, highlights the expression, then presses the **Numeric Enter** key. Note that the **Numeric Enter** key is located on the numeric keypad of your keyboard, and is labeled the same as the normal Enter key, but for our purposes it is distinct from the **Enter** key. The **Enter** key is used in the editor to insert a new line. It will not cause the Corman Lisp system to evaluate any text.

If your keyboard does not have a **Numeric Enter** key you may use **Shift-Enter**, i.e. hold down the shift key and press the normal **Enter** key, to get the same effect as **Numeric Enter**. You may find this method of executing Lisp code easier to use. In the rest of this document, we will refer to **Numeric Enter**, or **Shift-Enter**, as the **Execute** key.

Whenever the **Execute** key is pressed, Corman Lisp will execute some text from the worksheet or the currently active text window. To determine which text will be executed:

1. If any text is highlighted, that text will be executed.

2. If no text is highlighted, but the vertical bar cursor is located to the left of a left parenthesis or to the right of a right parenthesis, you should see a Lisp Selection. This is signified by corresponding left and right parentheses being underlined and colored red, to outline the current Lisp selection. Pressing **Execute** when a Lisp Selection exists will cause that text to be executed. The text being executed will momentarily flash to signify that it is being executed.

3. If neither of the above rules applies, the current line of text that contains the cursor will be executed. If it contains in incomplete Lisp expression, it may generate an error, or it may appear to hang, as it awaits further input to complete the expression.

Misc. Features

Open files, and their window positions, are now saved when the IDE is exited. They are automatically reopened and restored to their previous screen positions when the IDE is later restarted. This behavior may be controlled via the **Edit->Preferences** dialog.

The full path names of files are now displayed in the edit window title bars. This may be controlled in the **Edit->Preferences** dialog.

Syntax Coloring

Syntax coloring is more stable than in 2.5, and has more capabilities. It is turned on by default, but may be turned off in the **Edit->Preferences** menu. Note that even if you turn off syntax coloring, you can manually invoke it via the **Context** menu (right-click in an edit window).

Files with the following extensions will support syntax coloring:

- .lisp

- .cl

- .lis

The common lisp code which implements the syntax coloring algorithm may be found in the source file:

CormanLisp 3.0/sys/edit-window.lispIt is possible to modify this code to affect the behavior of the syntax coloring. There are also some variables (such as the colors used for various types of annotations) which may be modified in this file or in the startup file **init.lisp** (recommended).

You may enable or disable this feature in the **Edit->Preferences** dialog.

Context Menu

When you use the mouse to right-click on an edit window, a **Context** menu pops up. This **Context** menu contains actions which will act on any text selected. The code for the **Context** menu is in the source file:

```
CormanLisp 3.0/sys/context-menu.lisp
```

and is easily customizable. In this version, you can perform operations such as:

- Convert selected text format to pretty-printed format.

- Macroexpand and pretty-print the selected expression.

- Make Uppercase.

- Make Lowercase.

- Find the source file and line of the selected symbol (naming a function).

- Bring up the Hyperspec entry for a Common Lisp symbol.

- Go to a specific file and line number (select a line after calling CCL:FIND-IN-FILES).

- Add the selected variable or expression to the **Lisp Variables** panel in the toolbar (see below).

Lisp Variables Panel

By default, the right-hand panel in the toolbar contains the **Lisp Variables** panel. This can display an arbitrary number of Lisp special variable bindings (global level bindings, not thread re-bindings) and expressions. These are updated once each second. Since expressions get continuously executed, be careful not to add an expression that would cause side effects or take a significant amount of time to return.

By default, the **Lisp Variables** panel contains the current package, CL:*PACKAGE*. To add a variable or expression, select it and right-click, and select **Add to Lisp Variables Panel** from the **Context** menu.

If you wish to add items, you will probably want to move the panel to another location. If you drag it from the toolbar onto the main window, it becomes a floating panel which may be resized as necessary. You can also dock it on the left side of the IDE main windows, where it will expend to fill the left side of the window vertically.

Declarations Menu

The **Declarations** menu is a dynamic menu which parses the current Lisp edit window and lists all the declarations found in that file. The declarations will show up in the menu, and may be selected by clicking on the associated menu item. This will cause the edit window to reposition to the selected declaration and highlight the declaration. The declaration types are listed in the file `init.lisp`, and you may modify this list according to your preference.

Long menus may be traversed by typing the first character of the name of the declared object (function, macro, etc.).

Symbols Menu

The **Symbols** menu is a dynamic menu which contains a dynamic submenu for each package which currently exists in the system. Each submenu contains an entry which allows the user to switch to that package (IN-PACKAGE) and an entry for each external symbol exported by the package. Gliding over an item causes its tooltip information to be displayed in the status bar (the bottom of the main window). Selecting a symbol causes the symbol to be passed to the DESCRIBE operator, and the output of the DESCRIBE operator is output in the IDE Worksheet.

History Menu

This dynamic menu keeps track of recently executed expressions (executed via the top level Lisp loop) and allows you to easily view or reselect one of these expressions. Selecting an expression in the **History** menu causes the expression to be output to the Lisp Worksheet, and you may then execute it if you wish.

Tooltips

Tooltips are floating boxes that show up above or below a symbol when you glide over it with the mouse, or when you type in the symbol with the keyboard. Tooltips are intended to provide some useful information. The information provided is as follows:

If the currently selected or typed symbol is an external symbol in the Common Lisp package, its lambda list (according to the standard) is displayed.

Otherwise if the symbol names a function, macro or operator which is not an external Common Lisp symbol, but for which the system has a known lambda list, the lambda list is displayed. This includes functions that you may have written, internal system functions, etc. The lambda list often has descriptive names that help you understand what types of arguments to pass to the function or macro.

Otherwise if the symbol does not name a function, macro or operators, but does contain a global variable binding, the current global value of that variable is displayed.

8 Compiler

The compiler is always invoked by the evaluator to evaluate any expression. It takes either a lambda expression or a Lisp object as its input, and returns a function which, when executed, will effectively evaluate the passed object. If the passed object is a lambda expression, the compiler will return a compiled function. In the case of a symbol, the compiler will return a compiled function of no arguments which, when executed, will return the value of the symbol (or generate an error message if the symbol has no value). Other Lisp objects will produce a function which simply returns the object.

The compiler may be invoked in the standard way, via the Common Lisp `COMPILE` function, but this is of little value since all functions are always compiled. You may find it useful to explicitly invoke the internal function `COMPILE-FORM`, which will compile any arbitrary form and produce a function which, when executed, will evaluate that form. The function `COMPILE-FORM` is used by the evaluator internally. The function produced may be examined, using `DESCRIBE`, or disassembled using `DISASSEMBLE`. The evaluator essentially is implemented like this:

```
(defun eval (form) (funcall (cl::compile-form form)))
```

The Common Lisp `COMPILE-FILE` function is included with Corman Lisp 3.0 and is fully implemented. Compiled files (with the `. FASL` extension) are usually somewhat smaller and load faster than Common Lisp source files. If you wish to create compiled code for distribution, you may also use `SAVE-IMAGE` to create a new Lisp image. `LOAD-IMAGE` is typically the fastest way to load and execute compiled lisp code (although it overwrites any previously loaded files). See the Corman Lisp extensions section for information about `SAVE-IMAGE` and `LOAD-IMAGE` functions.

Declarations

The compiler was designed to generate very fast code without compromising safety or debug capabilities. With standard optimization settings (the defaults i.e. `SPEED 0`, `SAFETY 3`) you get many speed optimizations and full debug information. Instead of offering many ways to modify the compiler output, we have concentrated on trying to provide the best of all worlds at the standard settings. This includes detailed debug information (which is kept in a very compact form), argument type checking, and many optimizations which can be done quickly at compile time and which don't compromise debugging.

We understand that there is value in providing fine-grained control over compiler output, and we have provided some capabilities in this area. Look for more compiler options to be implemented in future versions of Corman Lisp.

INLINE/NOTINLINE

If the compiler encounters a PROCLAIM or DECLAIM form which includes an INLINE declaration for a given name, that name will be compiled inline whenever a call to the function is made. This requires that the function definition is available at the point the call is compiled. INLINE declarations at lexical scope are not currently supported.

A PROCLAIM or DECLAIM form with a NOTINLINE declaration will cancel any previously seen INLINE declaration. Note that NOTINLINE will not necessarily prevent system functions such as CAR or CDR from being inlined (there is not currently any way to prevent this automatically).

DYNAMIC-EXTENT

If a lambda &rest variable has been declared locally as having DYNAMIC-EXTENT, it will be created on the stack rather than on the heap. This can make use of &rest variables more efficient, but can also cause odd problems if you aren't very careful.

Example:

```
(defun foo (&rest args)
    (declare (dynamic-extent args))
    (apply 'list args))
```

TYPE

In many cases, the compiler will generate more efficient code if a type is known at compile time. This is particularly the case when a variable is known to contain a fixnum (small integer).

OPTIMIZE

If the optimize setting for SPEED is high and the optimize setting for SAFETY is low, then argument checking will be suppressed in function prologs.

SPECIAL

This is implemented according to the ANSI standard.

Technical Information

The compiler, when passed a lambda expression to be compiled, first examines the entire expression. If any macros are encountered, in an unquoted list, the macro expansion functions are called (repeatedly if necessary) until no macro calls remain. The compiler also looks for lambda expressions which are defined inside another lambda (and which may persist past the end of execution of the outer lambda) and looks for local variables which may be captured when the inner lambda closures are created at run time.

Code is generated, incorporating several optimizations. A number of Common Lisp functions are always inlined, such as CONS, EQ, =, <, >, +, -, CAR, CDR, etc. In addition, only those local variables which may possibly persist past the exit of the function call are included in the function environment. Other local variables are stack-allocated (or register allocated) only. This accounts for the vast majority of local variables in typical programs, and makes performance very good. When variables are captured by the environment, the generated code keeps the environment pointer in a register, making environment accesses nearly as fast as stack accesses.

The compiler may be extended using DEFCODEGEN. Whenever the compiler is about to compile an expression, it checks for a hook to a special code generation function. This may be used to extend the compiler behavior and implement things like special forms. See examples which use DEFCODEGEN for information about usage.

The compiler recognizes embedded assembler statements which use a special non-Lisp syntax. These may be used to access specific registers or do something that would otherwise not be possible in Common Lisp. You need to have a pretty deep understanding what you are doing before you make much use of this feature however. You may browse the Corman Lisp sources for examples of using inline assembler.

9 Assembler

There are times you need to use assembler, and nothing else will do. Most powerful programming language implementations include some way to hook into the power of the machine, and Corman Lisp is no exception. Several ways of embedding assembler instructions in your code are supported.

The following example shows how to create an entire function in assembler. There are many more examples of this in the Corman Lisp source file kernel-asm.lisp. To use assembler, you must understand the way Lisp objects are tagged (see the section on Lisp Data Structures).

DEFASM

Example:

```
;;;;
;;;;   Common Lisp IDENTITY function.
;;;;
(defasm identity (obj)
    {
        push    ebp                 ;; maintain stack frame
        mov     ebp, esp            ;; maintain stack frame
        cmp     ecx, 1              ;; number of args = 1?
        je      short :next         ;; if yes, branch
        callp   _wrong-number-of-args-error ;; else error
    :next
        mov     eax, [ebp + ARGS_OFFSET]   ;; eax = first arg
        mov     ecx, 1                     ;; returning 1 value
        pop     ebp                        ;; maintain stack frame
        ret
    })
```

DEFCODEGEN

Another use of assembler is to define special code generation code for the compiler, using `DEFCODEGEN`.

Example:

```
(defcodegen call-com-method (form dest)
    (let* ((vtable-index (cadr form)))
         (parse-assembler
          {
              mov eax, [esp]          ;; eax -> interface
              mov eax, [eax]          ;; eax -> vtable
              call [eax + (* 4 vtable-index)]
          }))))
```

Mixing Assembler With Common Lisp Code

A third method of embedding assembler into source code allows you to mix Lisp source and assembly code. Note the double curly braces surrounding the embedded assembly code.

Example:

```
(defun foo (x)
    (format t "Called foo...")
    {{
        mov    eax, [ebp + ARGS_OFFSET]
        mov    ecx, 1
    }})
```

Some assembler operators are not yet implemented, and not all addressing modes are supported. Only 8-bit and 32-bit operations are supported for most instructions, with `MOV` as the notable exception. You can browse the assembler source code (operators are defined with `DEFOP`) to see which operators are available.

DEFASM-MACRO

Assembler macros are supported via DEFASM-MACRO (in the X86 package).

Example:

```
(defasm-macro callf (sym)
    (let* ((env-offset (* (uref sym symbol-jump-table-offset) 4))
           (jump-offset    (+ env-offset 4)))
        (if (zerop env-offset)
            (error "The function ~A does not have a jump table offset" sym))
        {
            mov  edi, [esi + env-offset]
            call [esi + jump-offset]
        }))
```

10 Disassemble

A full Intel disassemble is included in Corman Lisp. You can disassemble any Corman Lisp function via the Common Lisp `DISASSEMBLE` function. Alternatively, you can disassemble bytes from any region of memory using the `DB:DISASSEMBLE-BYTES` function (`DEBUG` package).

The disassemble does not currently display any symbolic information, so it can be difficult to determine what functions are being called or variables are being referenced by disassembled code.

11 Debugging

Corman Lisp now includes a modal debugger—that is, a debugger loop that you can enter and do things which you can't do at the top level. Errors and calls to BREAK usually drop you into this debugger loop.

When in the debug loop, many keyword symbols have special meanings as debugger commands. You can execute the symbol :? at any time while in the debug loop to display a complete list of debugger commends.

While in the debug loop, at any given time a particular stack frame is considered active. Commands in the debugger allow you to move to a different stack frame, and to display the contents of all the variables in the current frame. In addition, while a frame is current, any lisp expressions you evaluate will implicitly refer to local variables in that frame, which means you can programmatically read and set these variables while in the debug loop.

You should be careful not to create persistent functions (such as with DEFUN) while in the debug loop, because you could inadvertently capture a local stack frame variable in the compiled function. If you later try to execute that function (after the stack frame is no longer valid) you would get unexpected (and potentially serious) results.

The Common Lisp TRACE facility is very useful, and always works for any function (since all functions are compiled, it handles compiled code). However, be very careful with Common Lisp functions. You generally should not TRACE them because they may be used internally by the system.

The Corman Lisp IDE makes it very easy to grab a copy of a function, add some FORMAT statements, execute the definition, and run. This gives customized debugging information in a few seconds, once you get fluent with the technique.

The DUMP-ERROR-STACK function, below, is useful, as are several other functions defined in the DEBUG (nickname DB) package.

DEBUGGER Commands

At the time of this writing, the following debugger commands are supported:

Command	Abbreviation	Explanation
`:CONTINUE`	`:C`	Exits the debug loop and invokes the specified restart option.
`:HELP`	`:?`	Displays this command list.
`:RESTARTS`	`:R`	Displays restart options.
`:FRAME`	`:F`	Print the current stack frame.
`:BACKTRACE`	`:B`	Print a backtrace down from the current frame.
`:NEXT`	`:N`	Go down the stack one frame.
`:PREVIOUS`	`:P`	Go up the stack one frame.
`:TOP`	`:<`	Go to the top of the stack.
`:BOTTOM`	`:>`	Go up the bottom of the stack.
`:GO`	`:G`	Go up to the specified frame.
`:LAMBDA`	`:L`	Display the lambda expression for the current function.

Debugging Functions

The following functions are exported by the DEBUG package. They may be used at any time (even when not in the DEBUG loop).

CONS-ADDRESS

CONS-ADDRESS *uvector*   *[function]*

Package: DEBUG

Returns the integer heap address of the passed CONS cell.

DISASSEMBLE-BYTES

DISASSEMBLE-BYTES *address num-bytes*   *[function]*
          *&optional (stream *standard-output*)*

Package: DEBUG

address should be an integer which represents a valid memory address.

num-bytes is an integer which represents the number of bytes to disassemble.

stream designates where to send the disassemble output.

DUMP-BYTES

DUMP-BYTES *obj*   *[function]*
          *&key (length *dump-bytes-default-length*)*
               *(stream *standard-output*)*
               *(width *dump-bytes-default-width*))*

Package: DEBUG

obj can be any lisp heap object or a positive integer which represents a valid memory address.

length, if specified, is the number of bytes to display.

stream defaults to *standard-output* and designates where to output the byte display.

width defaults to *dump-bytes-default-width* and designates the number of bytes to output on each line.

Causes *length* bytes to be displayed, starting at the heap address of the passed object, or at the memory address if an integer is passed.

DUMP-BYTES-DEFAULT-WIDTH   [*variable*]

Defaults to 8. This variable is used to control the number of bytes per line for
calls to DUMP-BYTES.

DUMP-BYTES-DEFAULT-LENGTH   [*variable*]

Defaults to 64. This variable is used as the default number of bytes to dump for
calls to DUMP-BYTES.

DUMP-DWORDS

DUMP-DWORDS *obj*   [*function*]
```
        &key (length *dump-dwords-default-length*)
             (stream *standard-output*)
             (width *dump-dwords-default-width*))
```

Package: DEBUG

obj can be any lisp heap object or a positive integer which represents a valid
memory address.

length, if specified, is the number of DWORDs (32-bit integers) to display.

stream defaults to *standard-output* and designates where to output the
byte display.

width defaults to *dump-dwords-default-width* and designates the
number of DWORDs to output on each line.

Causes *length* DWORDs to be displayed, starting at the heap address of the
passed object, or at the memory address if an integer is passed.

DUMP-DWORDS-DEFAULT-WIDTH   [*variable*]

Defaults to 1. This variable is used to control the number of DWORDs per line for
calls to DUMP-DWORDS.

DUMP-DWORDS-DEFAULT-LENGTH   [*variable*]

Defaults to 32. This variable is used as the default number of DWORDs to dump
for calls to DUMP-DWORDS.

DUMP-ERROR-STACK

DUMP-ERROR-STACK [*function*]

Package: DEBUG

Prints a trace of the stack which was captured when the last call to ERROR or BREAK was made. This tells you what functions were active, and with which arguments.

FIND-SOURCE

FIND-SOURCE *func-name* [*function*]

Package: DEBUG

func-name is a function or a symbol which names a function.

If you are running from the Corman Lisp IDE, this command opens an edit window which contains the source to the requested function, and positions the text cursor at the first line of that function. This only works if the function was defined in the process of a call to LOAD. The **Execute File...** menu command does not initialize this correctly. You may call FIND-SOURCE on Corman Lisp library functions if you like, to browse the source code.

PEEK-BYTE

PEEK-BYTE *address* [*function*]

Package: DEBUG

address should be an integer which represents a valid memory address.

Returns the byte which is stored at the passed memory address.

PEEK-DWORD

PEEK-DWORD *address* [*function*]

Package: DEBUG

address should be an integer which represents a valid memory address.

Returns the DWORD (32-bit integer) which is stored at the passed memory address.

STACK-TRACE

STACK-TRACE   [*function*]

Package: CORMANLISP

Returns a list representing the execution stack frames of the currently executing thread.

This function is used internally by ERROR to store the stack frame returned by DUMP-ERROR-STACK.

UVECTOR-ADDRESS

UVECTOR-ADDRESS *uvector*   [*function*]

Package: DEBUG

Returns the integer heap address of the passed uvector.

12 Memory Manager

The Corman Lisp Memory Manager consists of several functions to allocate memory, and the garbage collector. None of these functions are designed to be called explicitly. Instead, memory management code is called implicitly when you call Common Lisp functions.

Corman Lisp includes a state-of-the-art garbage collector for efficient use of memory. It is designed for maximum performance, no noticeable pauses, and good virtual memory behavior. Technically the Corman Lisp garbage collector (which is implemented in the kernel) has the following attributes:

- Copying (live data is moved during collection)

- Generational (there are several generations of heap objects sorted by age)

- Compacting (the collection process results in a compact heap with no fragmentation)

- Virtual-memory friendly (it works with the virtual memory system)

These attributes have been shown to be especially good for Common Lisp implementations. There are several implications of this, some positive, and some negative.

Memory allocation is extremely fast. It is nearly as fast as stack allocation. Since fragmentation is not an issue, the memory manager need only increment a pointer into free space to allocate a block of arbitrary size.

Garbage collection normally only collects Generation 0 data, and copies live data into Generation 1. This operation takes just a few milliseconds. When Generation 1 fills up, it causes live data from Generation 1 to be copied into Generation 2. The virtual memory hardware is used to detect when older generation heap objects have been modified to point to younger generation heap objects, thus avoiding the overhead of keeping track of that in software.

On the negative side, the fact that heap objects move around makes interfacing with other languages, such as C or C++, more of a challenge. Lisp heap objects which are exported to foreign functions must be referenced only indirectly, through pointers that the garbage collector can update. In addition, all data objects must be tagged with a few bits which indicate whether they are a heap pointer or some other type of data (such as an integer). Maintaining these tags actually has minimal impact on memory use and performance, but adds to the complexity of interfacing with foreign code.

Most Common Lisp functions which create Lisp data items call the memory allocator, and user code normally never calls the allocator functions explicitly. The function CONS allocates a 2-cell heap object, and functions such as VECTOR, MAKE-HASH-TABLE, and MAKE-INSTANCE all allocate heap objects.

The Common Lisp ROOM function may be used to determine how much heap memory is being used at any time. This function does not report memory being allocated and used by foreign functions or the operating system, only that used by Lisp heap objects.

There are several non-standard functions relating to memory management that you may wish to call in user code.

Memory Manager Functions

GC

GC *&optional (level 0)* *[function]*

Package: CCL

Explicitly invokes the garbage collector. The level may be in the range 0-3.

MAKE-WEAK-POINTER

MAKE-WEAK-POINTER *object1* *[function]*

Package: CCL

The object may be any Lisp heap object. This function returns a special object of type WEAK-PTR which contains one slot: the passed object. The reference is set up such that it will not cause the object to survive garbage collection. If no other pointers to the object exist, it will be collected when a normal garbage collection occurs. However, when that happens, the garbage collector will set the object slot of the weak pointer to NIL. This is a useful type of object for tracking outstanding instances of some data structure, for example.

WEAK-POINTER-P

WEAK-POINTER-P *objectl* [*function*]

Package: CCL

Returns T if the object is of type WEAK-PTR, otherwise it returns NIL.

WEAK-POINTER-OBJ

WEAK-POINTER-OBJ *weak-ptr* [*function*]

Package: CCL

The passed argument should be an object of type WEAK-PTR.

Returns the object that the weak pointer references.

REGISTER-FINALIZATION

REGISTER-FINALIZATION *object function* [*function*]

Package: CCL

object may be any Lisp heap object.

function should be a function which takes one argument.

This function instructs the garbage collector that when the passed object is collected, prior to collection the passed function should be executed. When the function is called, the garbage collector passes the object as its single parameter.

Implementation Note:

The implementation of REGISTER-FINALIZATION is such that the lifetime of the passed object is extended somewhat. The collector cannot know which objects will not survive collection until after a collection is finished (or at least underway). However, the finalization function must not be called during a collection (so as not to recursively cause garbage collection). As a result, the object to be finalized is *resurrected* after collection, its finalization function is called, the object is removed from the finalization list, and is then left to be collected at the next collection.

ENABLE-HARDWARE-GC

ENABLE-HARDWARE-GC &optional *enable?* [*function*]

Package: CCL

Enables or disables *hardware assisted* garbage collector mode.

HARDWARE-GC-ENABLED-P

HARDWARE-GC-ENABLED-P [*function*]

Package: CCL

Returns **T** if hardware assisted garbage collector mode is enabled. Returns **NIL** otherwise.

13 Foreign Function Interface

The Foreign Function Interface, or FFI, allows Common Lisp functions to easily call any functions which are exported by a Windows dynamic link library (DLL). It provides specific support for a C-language interface, through a package called `C-TYPES` (nickname `CT`). Using macros defined in `C-TYPES`, you may define C datatypes, including structures. You may want to look at example programs included with Corman Lisp which use these features. Any Win32 function may be called easily with this facility.

In addition to being able to use the FFI to call foreign functions, you may use the `DEFUN-CALLBACK` macro to define an interface to any Lisp function, which may be given to a foreign function for later calling. This is extremely useful in the Windows operating system calls, and is used for such things as Window procedures (WINPROCs).

Foreign Functions

DEFCTYPE

DEFCTYPE *name c type*   *[macro]*

Package: `C-TYPES`

name should be a symbol, and is not evaluated.

ctype is not evaluated, and should be a form which represents a c-type which is already defined.

Examples of predefined c-types are:

```
:void
:char
:unsigned-char
:short
:unsigned-short
:long
:unsigned-long
:short-bool
:long-bool
:single-float
:double-float
:handle
(:long 5)                ; array of 5 longs
(:long *)                ; pointer to long
((:long *) *)            ; pointer to pointer to long
(:struct f1 :short f2 :long)
                ; struct containing a short and a long
```

Example:

```
(ct:defctype string32 (:char 32)); declare a type which is a
                            ; C array of 32 characters
```

DEFWINTYPE

DEFWINTYPE *name ctype*   [*macro*]

Package: WIN32

This macro is like DEFCTYPE, except that it causes the defined type to be exported from the WIN32 package.

DEFCSTRUCT

DEFCSTRUCT *name field-definitions*   [*macro*]

Package: C-TYPES

This macro declares a C structure definition.

name should be a symbol (not evaluated).

field-definitions is not evaluated, and should be a list of field-definition descriptors, each of which consists of a symbol and a c-type descriptor.

Example:

```
(ct:defcstruct MEMORY_BASIC_INFORMATION
    ((BaseAddress        (:void *))
     (AllocationBase     (:void *))
     (AllocationProtect  :unsigned-long)
     (RegionSize         :unsigned-long)
     (State              :unsigned-long)
     (Protect            :unsigned-long)
     (Type               :unsigned-long)))
```

DEFWINSTRUCT

DEFWINSTRUCT *name field-definitions*   *[macro]*

Package: WIN32

This macro is like DEFCSTRUCT, except that it causes the defined structure type to be exported from the WIN32 package.

DEFUN-DLL

DEFUN-DLL *name param-list*   *[macro]*
   &key return-type library-name entry-name linkage-type

Package: C-TYPES

This macro causes a Lisp wrapper function to be built which, when executed, will call the specified DLL function.

name is a symbol which specifies the name of the Lisp wrapper function.

param-list is a list of parameter specifiers, each of which contains a variable name and a c-type specification.

return-type specifies the type of value returned by the function, if any. It defaults to :long.

library-name is a required key which specifies the name of the .DLL file which contains the function being called.

entry-name is the name of the .DLL function being called. It is normal that the *name* and *entry-name* function be the same, but this is not required.

linkage-type should be either :c or :pascal, with :c being the default. With :c linkage, the called function is not expected to remove the parameters from the stack when it returns. With :pascal linkage, it is assumed that the called function will remove the passed parameters.

DEFWINAPI

DEFWINAPI *name param-list*   *[macro]*
   &key return-type library-name entry-name linkage-type

Package: `WIN32`

This macro is like `DEFUN-DLL`, except that it causes the defined function to be exported from the `WIN32` package.

CREATE-FOREIGN-PTR

CREATE-FOREIGN-PTR   *[function]*

Package: `C-TYPES`

Returns a foreign pointer object, with the address slot initialized to 0.

SIZEOF

SIZEOF *c-type-specifier*   *[function]*

Package: `C-TYPES`

c-type-specifier is a valid C type specifier

Returns the number of bytes required to hold a C object of the passed type.

CPOINTER-VALUE

CPOINTER-VALUE *foreign-pointer*   *[function]*

Package: `C-TYPES`

Returns the foreign address stored in the passed foreign pointer.

You may use `SETF` to set this value.

CPOINTERP

CPOINTERP *object*   *[function]*

Package: `C-TYPES`

Returns `T` if the passed object is a foreign pointer, `NIL` otherwise.

DEFUN-CALLBACK

DEFUN-CALLBACK *name arglist &rest body*   *[macro]*

Package: `C-TYPES`

name is a symbol.

arglist is a list of the form:

```
((arg1-name  arg1-c-type)
 (arg2-name  arg2-c-type)
 etc.)
```

This macro defines a callback function which may be passed to a foreign function for later calling back into Lisp. The function that is created has `:pascal` linkage, which is the usual style for operating system callbacks in Windows. In other words, the callback function will remove its own parameters from the call stack.

Example:

```
(ct:defun-callback WndEnumProc
    ((hwnd HWND)
     (lParam LPARAM))
    (funcall *wnd-enum-proc* hwnd lParam)
    t)
```

DEFUN-C-CALLBACK

DEFUN-C-CALLBACK *name arglist &rest body*   *[macro]*

Package: `C-TYPES`

This macro is like `DEFUN-CALLBACK`, except that the defined callback function has `:c` linkage, i.e. the caller is expecting to remove the passed parameters from the stack.

GET-CALLBACK-PROCINST

GET-CALLBACK-PROCINST *callback-name*   *[function]*

Package: `C-TYPES`

callback-name is the name of a callback function defined with `DEFUN-CALLBACK` or `DEFUN-C-CALLBACK`.

This function returns the foreign pointer to the callback function, suitable for passing to foreign functions.

CREF

CREF *c-type object access* *[macro]*

Package: C-TYPES

c-type is a C type definition, and is not evaluated.

object is a foreign pointer to a C-style array or structure, and *is* evaluated.

access is an integer, in the case of an array, or a field name, in the case of a structure. If it is an array index, it is evaluated. If it is a field name it is not evaluated.

Example:

```
(ct:defcstruct point ((x :long)(y :long))) ; declare a C point structure
(setf p (ct:malloc (ct:sizeof 'point))) ; allocate a point
(setf (ct:cref point p x) 100)           ; set the point x value = 100
(setf (ct:cref point p y) 200)           ; set the point y-value = 200
(ct:cref point p x)  ->  100             ; access the point x-value

(ct:defctype point-array (point 10))        ;define a point array
(setf a (ct:malloc (ct:sizeof 'point-array))) ;allocate a point array
(dotimes (j 10)                          ; initialize each point
    (setf (ct:cref point (ct:cref point-array a j) x) j)
    (setf (ct:cref point (ct:cref point-array a j) y) j))
(ct:cref point (ct:cref point-array a 2) x) -> 2 ; retrieve p[2].x
```

C-STRING-LENGTH

C_STRING-LENGTH *c-string* [*function*]

Package: C-TYPES

c-string should be a foreign pointer which points to a 0-terminated character array.

Returns the number of characters in the array up to but not including the 0 terminating character.

CPOINTER=

CPOINTER= *pointer-1 pointer-2* [*function*]

Package: C-TYPES

pointer-1 and pointer-2 should be foreign pointers.
Returns T if the two pointers point to the same memory address, NIL otherwise.

CPOINTER-NULL

CPOINTER-NULL *pointer*   *[function]*

Package: C-TYPES

pointer should be a foreign pointer.

Returns T if the pointer is NULL, i.e. points to address 0.

MEMCMP

MEMCMP *p1 p2 count*   *[function]*

Package: C-TYPES

P1 and p2 should be foreign pointers.

count should be a fixnum.

Count pairs of bytes at p1 and p2 are compared. If the bytes at p1 and p2 are the same, 0 is returned. If a non-equal byte is encountered, if the lesser byte is in p1, -1 is returned, otherwise 1 is returned.

Returns:

```
-1 if p1 < p2
 0 if p1 == p2
 1 if p1 > p2
```

MALLOC

MALLOC *size*   *[function]*

Package: C-TYPES

size should be a fixnum.

Allocates a block of *size* bytes of memory from the foreign heap.

Returns a foreign heap pointer to the allocated block.

The block returned may be deallocated with FREE, but this is optional. If FREE is not called, the garbage collector will free the block when it collects the foreign pointer (via finalization).

FREE

FREE *pointer*   *[function]*

Package: C-TYPES

pointer should be a foreign heap pointer, allocated with MALLOC.

This function deallocates the block of foreign heap memory referenced by the pointer, and sets the pointer to 0.

Calling this function is optional, since the collector, when it frees the foreign heap pointer, will call FREE during finalization. It is useful if you are concerned about foreign heap size and you want to make sure the memory gets freed. Note that filling up the foreign heap never, in and of itself, causes Lisp garbage collection to occur.

FOREIGN-PTR-TO-INT

FOREIGN-PTR-TO-INT *pointer*   *[function]*

Package: C-TYPES

pointer should be a foreign pointer.

Returns the integer which represents the memory address the pointer is pointing to.

INT-TO-FOREIGN-PTR

INT-TO-FOREIGN-PTR *integer*   *[function]*

Package: C-TYPES

Returns a foreign pointer which has been initialized to point to the address the integer represents..

UNICODE-TO-LISP-STRING

UNICODE-TO-LISP-STRING *unicode-string*   *[function]*

Package: C-TYPES

unicode-string should be a foreign pointer to a null-terminated unicode string (16-bit characters).

Returns a newly created Lisp ASCII character string. If the unicode string contains characters where the high byte is non-zero, the characters will be truncated to 8 bits. This should be remedied in future releases of Corman Lisp, which will support Lisp strings of unicode characters.

LISP-STRING-TO-UNICODE

LISP-STRING-TO-UNICODE *string*   [*function*]

Package: C-TYPES

string should be a Lisp string.

Returns a newly allocated foreign heap pointer, which points to a newly allocated foreign heap block containing a unicode version (16-bit characters) of the passed Lisp string.

C-STRING-TO-LISP-STRING

C-STRING-TO-LISP-STRING *c-string*   [*function*]

Package: C-TYPES

c-string should be a foreign pointer to a null-terminated ASCII character string.

Returns a newly created Lisp string which contains the same characters as the passed c-string.

LISP-STRING-TO-C-STRING

LISP-STRING-TO-C-STRING *lisp-string*   [*function*]

Package: C-TYPES

lisp-string should be a Lisp string.

Returns a newly allocated foreign heap pointer, which points to a newly allocated foreign heap block containing an ASCII character string copy the passed Lisp string.

The returned ASCII string will have a 0 terminating byte.

C Function Definition Parser

The C declaration parser is written by <u>Vassili Bykov</u>, based on the original version by Roger Corman. It automatically generates foreign function interface (FFI) declarations from C source code.

The parser is invoked whenever a character sequence #! is encountered in Lisp code. This character sequence begins a *C declaration block*: a portion of the source containing declarations and preprocessor directives, supposedly pasted from a C header file. The declaration block is terminated by a declaration block end marker: !#. At the beginning of the block the parser expects to find a *translation parameter form*. The full syntax of parameter form is described <u>later in this text</u>. The C declarations in the declaration block are parsed, analyzed, and the <u>corresponding Corman Lisp FFI declaration forms</u> are generated.

As an introductory example, consider this C declaration block and the Lisp forms it is translated to.

```
#! (:library "User32" :ignore "WINUSERAPI" :pascal "WINAPI")

/* Just a few styles -- this comment is ignored, BTW */
// and this is ignored, too
#define WS_POPUP            0x80000000L
#define WS_BORDER           0x00800000L
#define WS_SYSMENU          0x00080000L
#define WS_POPUPWINDOW      (WS_POPUP          | \
                             WS_BORDER         | \
                             WS_SYSMENU)
typedef struct tagMSG {
    HWND        hwnd;
    UINT        message;
    WPARAM      wParam;
    LPARAM      lParam;
    DWORD       time;
    POINT       pt;
} MSG, *PMSG, NEAR *NPMSG, FAR *LPMSG;

WINUSERAPI BOOL WINAPI DrawEdge(HDC hdc, LPRECT qrc, UINT edge, UINT grfFlags);
!#
```

This block is translated into a Lisp form:

```
(PROGN
    (DEFWINCONSTANT WS_POPUP 2147483648)
    (DEFWINCONSTANT WS_BORDER 8388608)
    (DEFWINCONSTANT WS_SYSMENU 524288)
    (DEFWINCONSTANT WS_POPUPWINDOW
                        (LOGIOR (LOGIOR WS_POPUP WS_BORDER) WS_SYSMENU))
    (PROGN
        (DEFWINSTRUCT MSG   ((HWND HWND)
                             (MESSAGE UINT)
                             (WPARAM WPARAM)
                             (LPARAM LPARAM)
                             (TIME DWORD)
                             (PT POINT)))
        (DEFWINTYPE LPMSG (MSG *))
        (DEFWINTYPE NPMSG (MSG *))
        (DEFWINTYPE PMSG (MSG *)))
    (DEFUN-DLL DRAWEDGE
            ((HDC HDC) (QRC LPRECT) (EDGE UINT) (GRFFLAGS UINT))
        :RETURN-TYPE BOOL
        :LIBRARY-NAME "User32.dll"
        :ENTRY-NAME "DrawEdge"
        :LINKAGE-TYPE :PASCAL))
```

Note how the translation parameters affected the translation: the `:library` parameter set the name of the library used to call the API defined in the block. The `:ignore` parameter instructed the translator to ignore the WINUSERAPI declaration, which would otherwise be considered to be a part of the function return type declaration.

Translation Parameters

The parameter form is a list of the following syntax:

```
(&key library export ignore pascal translate trim-last auto-ansi
verbose)
```

The mode parameter is required while the others are optional with the exception of library, which is required if the translation block contains function prototypes.

library

This parameter value should be a string. It names the DLL where the functions defined in this declaration block are to be found. If the name does not have an extension, ".dll" is added. This parameter is required when the block contains function declarations. In other cases it is optional.

export

This parameter value is a generalized boolean that indicates whether the names of all constants, types, and functions defined as the result of translating the block should be exported from the defining package. The default value is nil.

ignore

This parameter should be a string or a list of strings (or symbols, in which case their names are used). All tokens in function prototypes in the block that match one of the strings are ignored. This is useful for such declarations as WINUSERAPI in WINUSER.H or WINCOMMCTRLAPI in COMMCTL.H, used in C to specify functions as exported from a DLL. Such declarations are of no importance on the Lisp side. Because they are usually attached to every function prototype in a .h file, it is easier to specify them as ignored than edit the function prototypes manually to remove them. In other words, consider :ignore "WINUSERAPI" to be the Lisp equivalent of #define WINUSERAPI in a C header file.

pascal

This parameter should be a string or a list of strings (or symbols, in which case their names are used). Functions that have one of these tokens in their prototype will have :pascal linkage, other functions will have the default :c linkage. Very often, working with Win32 declarations, this parameter would have "WINAPI" as its value.

translate
trim-last
auto-ansi

These three parameters control name translation and are described in the [Name Translation](#) section.

verbose

When true, the C definitions and their translations are printed on the *standard-output* during translation. The default value is NIL.

FFI Definitions

The following table gives an idea of what C declarations are translated to.

C	Lisp
`#define <name <expr`	`(defconstant <name <translated-expr)`
`typedef int *pint;`	`(defctype pint (:long *))`
`typedef int *pintarr[5];`	`(defctype pintarr ((:long *) 5))`

`typedef struct {`
`  int x;`
`  pint y;`
`} FOO, *LPFOO;`
	`(defcstruct foo ((x :long) (y :pint)))`
	`(defctype lpfoo (foo *))`

`struct IDirectDraw;`
`typedef struct IDirectDraw`
`    *LPIDirectDraw;`
	`(defcstruct IDirectDraw nil)`
	`(defctype LPIDirectDraw (IDirectDraw *))`

`WINUSERAPI BOOL WINAPI`
`ShowWindow (HWND hWnd, int nCmdShow);`
	`(defun-dll showwindow`
	`    ((hwnd hwnd) (ncmdshow :long))`
	`  :return-type bool`
	`  :entry-name "ShowWindow"`
	`  :library "User32.dll"`
	`  :linkage-type :pascal)`

`interface {...};`	See COM section.

All definitions occur in the package where the C declaration block is. The name of the defined entity is obtained from the corresponding C name. If case insensitivity is a problem and two names collide, the `:translate` parameter can be used to rename one of them.

By default, the defined symbols are not exported. They may be exported individually, but `:export` translation parameter allows to export all the defined names.

Name Translation

In some cases, control over name translation is required. It is provided by three translation parameters. In order of increasing power (and decreasing convenience) they are `:auto-ansi`, `:ansi:`, and `:translate`.

auto-ansi

This parameter is a generalized boolean. When true, the parser translates any function name that looks like an ANSI-specific version of the "actual" function (that is, the last character is an uppercase "A" while the preceding character is lowercase), to a Lisp name with the last character thrown away.

This parameter is true by default, which means that "CreateWindowA", for example, produces Lisp function CREATEWINDOW still using "CreateWindowA" as the DLL entry point name.

Note that this translation is only applied to function names. Types and constants are typically all uppercase and the rule used to determine if the name is an ANSI-specific version would not work for them. To avoid global edits to change all `CREATESTRUCTA` to `CREATESTRUCT` in a large body of text, the following parameter is useful.

trim-last

The value of this parameter should be a string or a list of strings. Whenever the parser translates a C name to a Lisp name, it looks up the name in this list (the lookup is case-sensitive). If found, the parser trims the last character of the name to get the name of the Lisp symbol to use.

For example, if you specify `:trim-last "LPCREATESTRUCTA"` in a C block header, this name will be translated to `LPCREATESTRUCT` wherever it is mentioned in the block, be it a defined type name, or a function parameter type.

translate

If even more control over name translation is required, this parameter allows to specify arbitrary translations. The value of it should be a list of two-element lists. The first element of each list is a string specifying a C name, the second element is a symbol to use as the translation of that name. For example, the following could be used to resolve name conflict:

```
:translate (("foo" small-foo) ("FOO" big-foo))
```

Supported C Syntax

Typically, in order to import C declarations, you would copy a block of declarations from an .H file and paste it into a declaration block in a .lisp file. However, the parser supports only a subset of C syntax, therefore it is essential to understand the differences between the "regular" C and the language understood by the parser. Some of the copied declarations would have to be removed, others slightly edited. In practice, however, the majority of declarations can be used "as is". In any event, tweaking C declarations to fit the parser limitations is much easier than writing the corresponding Lisp FFI declarations by hand.

Case Sensitivity

Each declared C name is translated into a definition associated with a Lisp symbol. By default, symbol naming in Lisp is case-insensitive. This means that declarations for "foo" and "FOO" would clash when translated to Lisp. This is unlikely to often be a concern, but if it is, `:translate` parameter can be used to specify an alternative translation for a name.

Packages

All symbols for the C definitions are created in the package current at the time the parser was invoked. If a declaration refers to other declaration names, the symbols for those names should be visible, or an error will be signaled by the FFI. For instance, the declaration block at the beginning of this document can only be translated somewhere where the symbols from the "WIN32" package for names such as "HDC" or "LPARAM" are visible.

#define semantics

In C, `#define` defines token substitution. As far as this parser is concerned, `#define` defines a constant. This means that the parser expects a valid expression on the right hand side, and the expression is evaluated to produce a value which is assigned to the constant. This means that some cases, such as

```
#define TIMES2          *2
#define TWOHUNDRED      100 TIMES2
```

are legal in C but not legal for this parser (it does not perform token substitution), other cases, such as

```
#define FOO      3 + 4
#define BAR      (FOO * 6)
```

are interpreted differently. For a C preprocessor, BAR is '3 + 4 * 6', and the effective value used in place of B somewhere in C would be 27. For this parser, BAR is (* FOO 6), and FOO is (+ 3 4), so the effective value of BAR would be 72.
The expressions are supposed to be arithmetic only; that is, the terms may only be identifiers, literal numbers, or literal strings.

The following operators are supported (in order of decreasing priority):

- ~	(Unary) negation and bitwise complement.
* / %	Multiplication, division, and modulus.
+ -	Addition and subtraction.
&	Bitwise AND.
^	Bitwise XOR.
|	Bitwise OR.

Since #define is not token substitution for this parser, #define with arguments is not allowed.

Some other preprocessor directives are recognized, namely: #if, #ifdef, #ifndef, #else, #endif. They are ignored with a warning. (A warning is needed because when these declarations are ignored, incorrect definitions may be included in the resulting translation).

#LISP escape
In some cases, a Lisp construct should accompany the body of C definitions. One of the cases when this may be required is when #define is used to declare a macro with parameters, not supported by the C declaration parser. In this case, a Lisp function or macro should be defined equivalent to the original C macro. While it is possible to add such a definition before or after the C declaration block, it is often convenient to place it in the C declaration block, next to the commented-out original C declaration. The #LISP escape makes this possible. If a keyword #LISP is encountered in a C declaration block, the form following it is read by the regular Lisp reader and the result is embedded into the sequence of FFI statements produced as the result of translating the C declaration block. Here is an example taken from the declaration block of ODBC 3.0 interface:

```
/* test for SQL_SUCCESS or SQL_SUCCESS_WITH_INFO */
// #define SQL_SUCCEEDED(rc) (((rc)&(~1))==0)
#LISP (progn
        (export 'SQL_SUCCEEDED)
        (defun SQL_SUCCEEDED (rc) (zerop (logandc2 rc 1))))
```

Type Definitions

Only type declarations matching the following template are supported:

```
typedef <type <pointers <name <dimension?
<pointers ::= '*'*
<dimension ::= '[' <integer | <identifier ']'
```

In other words, no pointers to functions, parentheses to specify priority, etc. More complicated types have to be broken into simple parts that fit the above pattern and gradually defined in terms of each other.

Once again, because #define does not define token substitution, the following declaration is not supported:

```
#define ASIZE 10
typedef int *pintarray[ASIZE];
```

The size of the array must be an integer or an identifier. In this case the identifier should be a constant defined either earlier in the same C declaration block, or existing in the system by the time the declaration block's translation is evaluated.

These limitations also apply to other type declarations: declarations of struct members, struct types (the part of typedef struct following the closing brace), function (including COM methods) parameter and return types.

Structure Definitions

Both struct and typedef struct are acceptable.

In the former case, the structure tag is used as the type being defined, while variable declarations following the structure body are ignored.

In the latter case, if a field list is supplied, the structure tag is ignored. The variable declarations following the struct body are all used to define the related types. Any typedef struct should define at least one *primary type name*—the name of the structure type itself. All other related types are defined in terms of the primary type. In the above example, MSG is the primary type, while LPMSG and others are defined in terms of it. An error is signaled if there is no primary type in a typedef struct statement.

If there is no field list supplied in a "typedef struct" declaration, the structure tag is assumed to be defined earlier using "struct" declaration. The types declared in the declaration are then declared based on the structure tag, as shown in the example above.

In the struct body and the variable list, the limitations described in the Type Definitions section apply.

Function Definitions

Most function prototypes typically found in Windows .H files are acceptable. The limitations on argument declaration complexity described in [Type Definitions](#) section apply.

pascal, or _pascal keywords in the function prototype cause the function to use :pascal linkage. The function will also use :pascal linkage if its prototype contains one of the tokens specified in the :pascal keyword parameter in the block's translation parameters. Often, with Win32 USER module declarations such a parameter would be used to specify "WINAPI" as a pascal linkage selector. If none of the above is the case, a function uses **:c** linkage.

The DLL entry name is always the one matching the function name as it appears in the C declaration. The name of the corresponding Lisp function can be different if :auto-translate-ansi or :ansi or **:translate** settings are used (see translation parameters description). As an example,

```
#! (:library "User32"
    :ignore "WINUSERAPI"
    :pascal "WINAPI"
    :trim-last "WNDCLASSA")
WINUSERAPI ATOM WINAPI RegisterClassA( CONST WNDCLASSA *lpWndClass);
!#
```

produces Lisp definition:

```
(DEFWINAPI REGISTERCLASS ((LPWNDCLASS (WNDCLASS *)))
    :RETURN-TYPE ATOM
    :LIBRARY-NAME "User32.dll"
    :ENTRY-NAME "RegisterClassA"
    :LINKAGE-TYPE :PASCAL)
```

The function name has been auto-translated because this is the default setting in the Win32 translation mode, the WNDCLASSA type name is translated to WNDCLASS because it is explicitly declared as ANSI-mangled.

COM Interfaces

The definitions from the Corman Lisp examples:

```
interface IUnknown {
    HRESULT QueryInterface(REFIID riid, LPVOID* ppvObject);
    ULONG AddRef();
    ULONG Release();
};
interface IMalloc : IUnknown {
    LPVOID Alloc(ULONG cb);
    LPVOID Realloc(LPVOID pv, ULONG cb);
    VOID Free(LPVOID pv);
    ULONG GetSize(LPVOID pv);
    int DidAlloc(LPVOID pv);
    VOID HeapMinimize();
};
```

are translated to the following Lisp form. Note that extra care is taken to make sure interfaces use method lists of the parent interfaces at the evaluation time as opposed to the method list in the time of macro expansion.

```
(PROGN
    (LET ((#:G851 0))
        (DEFUN-COM-METHOD QUERYINTERFACE-IUNKNOWN
                  ((WIN32:INTERFACE *)
                   (RIID REFIID)
                   (PPVOBJECT (LPVOID *)))
             (+ #:G851 0)
             :RETURN-TYPE HRESULT)
        (DEFUN-COM-METHOD ADDREF-IUNKNOWN ((WIN32:INTERFACE *))
            (+ #:G851 1)
            :RETURN-TYPE ULONG)
        (DEFUN-COM-METHOD RELEASE-IUNKNOWN ((WIN32:INTERFACE *))
            (+ #:G851 2)
            :RETURN-TYPE ULONG)
        (SETF (INTERFACE-METHOD-LIST 'IUNKNOWN)
            (APPEND NIL '(QUERYINTERFACE-IUNKNOWN
                          ADDREF-IUNKNOWN
                          RELEASE-IUNKNOWN))))
    (LET ((#:G852 (LENGTH (INTERFACE-METHOD-LIST 'IUNKNOWN))))
        (DEFUN-COM-METHOD ALLOC-IMALLOC ((WIN32:INTERFACE *) (CB ULONG))
            (+ #:G852 0)
            :RETURN-TYPE LPVOID)
        (DEFUN-COM-METHOD REALLOC-IMALLOC ((WIN32:INTERFACE *)
                                            (PV LPVOID)
                                            (CB ULONG))
            (+ #:G852 1)
            :RETURN-TYPE LPVOID)
        (DEFUN-COM-METHOD FREE-IMALLOC ((WIN32:INTERFACE *) (PV LPVOID))
            (+ #:G852 2)
            :RETURN-TYPE VOID)
        (DEFUN-COM-METHOD GETSIZE-IMALLOC ((WIN32:INTERFACE *) (PV LPVOID))
            (+ #:G852 3)
            :RETURN-TYPE ULONG)
```

```
        (DEFUN-COM-METHOD DIDALLOC-IMALLOC ((WIN32:INTERFACE *) (PV LPVOID))
            (+ #:G852 4)
            :RETURN-TYPE :LONG)
        (DEFUN-COM-METHOD HEAPMINIMIZE-IMALLOC ((WIN32:INTERFACE *))
            (+ #:G852 5)
            :RETURN-TYPE VOID)
        (SETF (INTERFACE-METHOD-LIST 'IMALLOC)
            (APPEND (INTERFACE-METHOD-LIST 'IUNKNOWN)
                    '(ALLOC-IMALLOC
                      REALLOC-IMALLOC
                      FREE-IMALLOC
                      GETSIZE-IMALLOC
                      DIDALLOC-IMALLOC
                      HEAPMINIMIZE-IMALLOC)))))
```

Transcribing Lisp Files

It is possible to convert a file containing Lisp forms and C declaration blocks into a
file containing only Lisp forms, with all C declarations replaced with their Corman
Lisp FFI equivalents. This operation is performed by the following function.

transcribe-file in-file out-file &optional (package *package*)
(prettyp t)

Opens `in-file`, reads all forms in it and writes them into `out-file`. (Both
parameters are strings). The output file will have Lisp FFI declarations expanded in
place of C declarations. The original formatting and the comments are, of course,
lost. The optional `prettyp` parameter, a generalized boolean, determines whether
the output file forms are printed prettily or not.

Transcribing the file eliminates the need for C parsing when the file is loaded into
Lisp, while Lisp FFI forms are easier to parse than the original C declarations. The
speed gain, depending on the file, may range from 5 to 50%. Care should be taken,
however, to ensure that a proper package is used while the file is being transcribed,
or the symbols printed in the output file may appear to be interned incorrectly when
the output file is loaded. To control this, optional `package` parameter (a package
designator) may be used.

14 ANSI Common Lisp Compatibility

Corman Lisp has been developed according to the ANSI Common Lisp specification. However, the specification is very large, and it is unfortunately not possible to offer a 100% ANSI compliant implementation at this time. The vast majority of ANSI functions are included, all the data types, and the majority of the specified functionality and behaviors. Any deficiencies regarding ANSI compatibility will be addressed in future releases. In the 3.0 release, a lot of progress has been made toward complete standard compliance.

CLOS

The CLOS implementation in Corman Lisp was originally based on the Closette implementation in the book *The Art of the Metaobject Protocol* by Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. The source to this implementation is included (in the file `clos.lisp` in the `library` folder). This implementation of CLOS is a subset of the full ANSI CLOS specification, and included non-ANSI extensions which provide a metaobject protocol (MOP). The following limitations apply (according to Chapter 1, section 1.1 of the book):

> All the essential features of full CLOS are included: *classes*, which inherit structure and behavior from one or more classes, *instances* of classes, which are created, initialized and manipulated; *generic functions*, whose behavior depends on the classes of the arguments supplied to them; and *methods* which define the class-specific behavior and operations of generic functions. The major restrictions of the simplified dialect include:

> **No class redefinition.** Full CLOS allows the definition of a class to be changed; the changes are propagated to its subclasses and to extant instances. The subset does not allow classes to be redefined.

> [Corman Lisp modification: This restriction has been removed. Classes may be redefined, but existing instances are not properly updated as specified by ANSI CL.]

> **No method redefinition.** Full CLOS allows methods to be redefined, with the new definition completely replacing the old one. The subset does not allow methods to be redefined.

> [Corman Lisp modification: This restriction has been removed. Methods may be redefined.]

> **No forward-referenced superclasses.** Full CLOS allows classes to be references before they are defined. One class can be defined in terms of another before the second has been defined. These forward references are not permitted in the subset.

> **Explicit generic function definitions.** Full CLOS allows the definition of a generic function to be inferred from the method definitions. The subset requires that a generic function be explicitly introduced with a `defgeneric` form before any methods are defined on it.

[Corman Lisp modification: This restriction has been removed. DEFMETHOD forms do not require a preceding DEFGENERIC.]

Standard method combination only. Full CLOS provides a powerful mechanism for user control of method combination. The subset defines only simple "demon" combination (primary, before- and after- methods).

No `eql` specializers. Full CLOS allows methods to be specialized not only to classes, but also to individual objects. The subset restricts method specialization to classes.

[Corman Lisp modification: EQL specifiers are now supported.]

No slots with :class allocation. Full CLOS supports slots allocated in each instance of a class, and slots which are shared across all of them. The subset defines only per-instance slots.

Types and classes not fully integrated. Full CLOS closely integrates Common Lisp types and CLOS classes. It is possible to define methods specialized to primitive classes (e.g. `symbol`) and structure classes (defined with `defstruct`). The subset defines classes for the primitive Common Lisp types but not for structure classes.

[Corman Lisp modification: Structures have unique classes and may be used to specialize methods.]

Minimal syntactic sugar. A number of convenience macros and special forms are not included in the subset. These include `with-slots`, `generic function`, `generic-flet` and `generic-labels`.

[Corman Lisp modification: WITH-SLOTS is implemented, courtesy of Vassili Bykov.]

The book is highly recommended if you want to work with CLOS. The file `clos-tests.lisp,` included with Corman Lisp in the examples folder, contains most of the code samples from the book, and is useful for seeing how CLOS works.

In Corman Lisp, the original Closette has been modified and rewritten extensively, especially with regard to generic function invocation. This has been done to improve performance, which is now substantially better than in previous versions.

15 Conditions

Much of the code to implement the ANSI condition system is in place (and included) it is not yet integrated with the Common Lisp library functions i.e. few functions signal conditions at this time. Work on this is ongoing.

16 Profiler

Profiler Functions

PROFILING

PROFILING *(funcs) &rest body*   *[macro]*

Package: `CCL`

funcs should be a list of function names to be profiled. These are not evaluated.

body is a list of forms to be evaluated. These form an implicit `PROGN` to be evaluated while profiling the functions specified.

The profiler timings will be inflated somewhat, because turning profiling on slows down execution. It is mainly useful for determining relative execution timings. Also, recursive functions will have their timings overstated, because the timings reflect the sum of times of all the invocations of a function. The timing is the time from when a function is entered to the time it is exited. It includes time spent in functions that it calls.

After the profiled forms are executed, a listing will be printed of how much time, in seconds, was spent in each profiled function. The profiler timings will be inflated somewhat, because turning profiling on slows down execution. It is mainly useful for determining relative execution timings. Also, recursive functions will have their timings overstated, because the timings reflect the sum of times of all the invocations of a function. The timing is the time from when a function is entered to the time it is exited. It includes time spent in functions that it calls.

17 Creating Standalone Applications

Application Functions

SAVE-APPLICATION

SAVE-APPLICATION *application-name start-function*   *[function]*
        &key (console nil) (static nil)
Package: `CCL`

application-name should the name of the application to be created. If the name does not end with a .EXE extension, one will be added automatically.

This function will create an executable in the **output directory** if absolute executable pathname was not given *(See Section 6, Subsection "User Specific Files and Directories" for details).*

start-function is the function which will be invoked as the top-level function in the saved application. It should do any necessary initialization, and launch the application.

console defaults to `NIL.` If true, it will create a standalone console application. If `NIL`, it will create a Windows application. A Windows application is responsible for all its own input and output. No edit window for receiving or displaying text will be provided.

static defaults to `NIL`. If true, the CormanLispServer DLL will be statically linked to the resulting application. This allows the application to be distributed as a single file, without including a separate DLL. It also prevents a different version of the CormanLispServer from inadvertently causing a problem later. If *static* is nil, the CormanLispServer DLL is required, but the resulting application will be smaller by around 300k bytes.

Note that the saved lisp code is now stored directly in the .exe file, and a separate .IMG file is not created or required.

Use `SAVE-APPLICATION` to create a binary application which can be launched from Windows just like any other application. Before using this function, you should do the following steps:

4. Start Corman Lisp.

5. Load exactly what you need to execute your application, using the Lisp `LOAD` command or the **Execute File…** menu command.

6. Call `SAVE-APPLICATION` with appropriate parameters.

   If you want to create another application, you should exit Corman Lisp and follow the above steps again. Everything that you do in Corman Lisp, prior to executing `SAVE-APPLICATION`, will affect the heap image and the saved application, so it important to start fresh each time.

   `SAVE-APPLICATION` actually does the following things:

   It creates a clone of one of the supplied bootstrapping applications, clboot.exe, clbootapp.exe or clconsole.exe.

   If the console key parameter is NIL it uses clboot.exe, otherwise it will use clconsole.exe. The cloned application is very small, and will have the name of the application you are creating.

   The variable `*top-level*` is set to the passed start-function.

   `SAVE-IMAGE` is called to create a copy of the lisp heap in the .EXE file in a segment of that file called `.lisp`.

   You launch the resulting application by double-clicking on the application file with the .EXE extension. When you distribute this application, you must distribute either one or two files, depending whether *static* was true (assuming your application is named *appname*):

   ```
   Appname.EXE
   CormanLispServer.DLL (if you did not use :static t)
   ```

   The `CormanLispServer.DLL` is shared by all applications which are built with Corman Lisp. It is a registered COM server, and only one instance of it is required on a given workstation.

18 Creating DLLs

You can create DLLs in Corman Lisp which may be called from any Windows applications. Since most application plug-in architectures are based on DLLs, this allows Corman Lisp code to be used to create plug-ins and extensions for a variety of applications. It also allows you to conveniently use lisp functionality within a program written in another language.

This section describes the function, COMPILE-DLL, which you use to create a DLL. It also explains how you can build and run an example program. Finally, technical information regarding the DLL runtime model is provided. For further information about DLLs, refer to "Microsoft Portable Executable and Common Object File Format Specification", which is available from Microsoft.

COMPILE-DLL

COMPILE-DLL input-file   *[function]*
          &key output-file
                (verbose *compile-verbose*)
                (print *compile-print*)
                def
                h
                (kernel "CormanLispServer.dll")
                (image "CormanLisp.img")
Package: `CCL`

output-file is the name of the .lisp file which you wish to compile as a DLL.
Functions defined with the macros `defun-dll-export-function` and
`defun-dll-export-c-function` will be exported from the DLL.

NOTE: The DLL will be created in the ***output directory*** if absolute DLL
pathname was not given *(See Section 6, Subsection "User Specific Files and Directories"
for details)*.

verbose defaults to `NIL`. If true, causes extra information to be output to
`*standard-output*`. This option gets passed on to `compile-file`.

print defaults to `NIL`. If true, causes extra information to be output to
`*standard-output*`.

This option gets passed on to `compile-file`.

def defaults to `NIL`. If true, creates a `.def` file for use by the linker when other
applications are linked to this DLL. It will be created in the same directory as the
output DLL.

h defaults to `NIL`. If true, automatically creates a .h file which may be included
in a C or C++ program which is designed to link to the DLL. It will be created
in the same directory as the output DLL.

`kernel` defaults to `CormanLispServer.dll`. The DLL, when loaded,
detects whether the Corman Lisp kernel is loaded. If not, it loads it. This option
allows a path other than the default to be specified for this.

Note: It is possible to copy the kernel DLL, and save a lisp image to the copied
DLL (using `SAVE-IMAGE`). This allows the kernel and lisp image to reside in the
same file. In this case, specify the same name for both the *kernel* and *image*
keyword options.

image defaults to CormanLisp.img. If the Corman Lisp kernel needs to be loaded by the DLL (if not already running) this option specifies the name of the Lisp image to load when the kernel starts. See the note above, regarding the possibility of combining these two files.

When your DLL is distributed, both the kernel and lisp image files must accompany the distributed DLL for proper execution. Your right to distribute these files is subject to the terms and conditions in the Corman Lisp license agreement.

DEFUN-DLL-EXPORT-FUNCTION

DEFUN-DLL-EXPORT-C-FUNCTION

DEFUN-DLL-EXPORT-FUNCTION *name arglist*   [*macro*]
        &optional prototype
        &rest forms

DEFUN-DLL-EXPORT-C-FUNCTION *name arglist*   [*macro*]
        &optional prototype
        &rest forms

Package: C-TYPES

These macros are used to define functions to be exported by a DLL (using COMPILE-DLL). If DEFUN-DLL-EXPORT-FUNCTION is used, the exported function will use Windows stdcall linkage. If DEFUN-DLL-EXPORT-C-FUNCTION is used, the exported function will use C linkage. DLLs used by C programs typically have C linkage, and Windows operating system DLLs typically use stdcall linkage. It doesn't matter which you use, as long as any applications which call functions exported by your DLL use the proper declarations that correspond with the linkage type.

name is a symbol or a list. If a symbol, the symbol names the lisp name for the function that will be defined as the export function, and its SYMBOL-NAME is used as the exported name in the DLL. If you wish the exported name to differ from the SYMBOL-NAME, then a list can be specified consisting of two items: the symbol used as the lisp name, and a string representing the DLL export name.

arglist is a list of the form:

```
((arg1-name  arg1-c-type)
 (arg2-name  arg2-c-type)
 etc.)
```

prototype if supplied, should be a string which is used to build the C include file (.h file) used to link to the function being defined by foreign code.

Example:

```
(ct:defun-dll-export-c-function (lisp-single-add "lisp_single_add")
    ((x :single-float)(y :single-float))
    "float lisp_single_add(float a, float b)"
    (+ x y))
```

Exporting Functions Using __stdcall Conventions

If you wish to export functions using __stdcall calling convention
(sometimes known as WINAPI), you need to modify the exported function
name using Microsoft's standard name-mangling scheme. In this case we add an
@ character followed by the number of bytes that will be passed to the function.
This should be 4 bytes for each argument, or 8 bytes of each double-float
argument. Here is an example:

```
(ct:defun-dll-export-function (lisp_add "lisp_add@8")
    ((x :long)(y :long))
    "long __stdcall lisp_add(long a, long b)"
    (+ x y))
```

This function is called by the name lisp_add from C or C++, and uses
__stdcall calling conventions.

Examples

DLLSample Example

To build the DLLSample example perform the following steps.

```
(ccl::compile-dll "examples/dllsample.lisp"
    :output-file "dllsample.dll"
    :verbose t
    :print t
    :def t
    :h t)
```

This compiles the sample file dllsample.lisp, and builds the output DLL
dllsample.dll. It also creates the .def file dllsample.def, and the C
include file dllsample.h.

```
(win::system "lib /def:dllsample.def /out:dllsample.lib /machine:x86")
```

This command runs the LIB command to create an import library for load-time linking of a client application to the DLL. The import library is called `dllsample.lib`.

Note: The last step requires that you have a copy of the Microsoft LIB utility installed on your workstation.

Building the DLLClient Example

To build the DLLClient example, you must have Microsoft Visual Studio.NET installed on your computer. To use a different compiler you may need to modify these steps and/or the makefile.

Open a command window, and go to the examples\dllclient directory.

Execute the following command:

```
nmake
```

This will build the program, and create the file dllclient.exe in the Corman Lisp root directory.

Run this program from the command line by executing:

```
dllclient
```

This should run the command line example program, which loads and runs the Corman Lisp kernel and image as part of initializing the `dllsample` DLL.

Multiple DLLs

Multiple DLLs may be built using `COMPILE-DLL`, and they may all be used at the same time by a client application (within the same process). However, any DLLs which run in the same process must share the same kernel and image file names, and be designed to share the same image file. You cannot necessarily determine which DLL will load first, and whichever one loads first will load the kernel and lisp image files that were specified when it was compiled. Only one lisp image file may be loaded at a time.

Limitations

Up to 100 exported functions may be defined in a DLL built using `COMPILE-DLL`. Multiple DLLs may be used to support >100 functions.

Technical Information

The DLLs that are built using Corman Lisp have initialization code which performs the following functions:

- It determines whether the Corman Lisp kernel is loaded and active in the process. If not, it loads it using the name of the kernel and lisp image file defined when the DLL was built with `COMPILE-DLL`.

- Once the kernel is loaded, the `FASL` segment of the DLL is loaded by the kernel. This contains the compiled code from the input lisp file when `COMPILE-DLL` was executed. All the functions and data from that file are loaded into the lisp image.

- The calling thread is initialized as a valid lisp thread, using the `DirectCall` function `BlessThread()`.

- Foreign entry points to all the exported functions are initialized, allowing subsequent client direct calls to the exported lisp functions.

DLLs which are built from unlicensed (evaluation) versions of Corman Lisp contain code that will execute occasionally when a DLL is loaded. This code displays a modal dialog informing the person running the client application that a DLL was created with an evaluation version of Corman Lisp, and it is not intended for distribution.

19 DirectCall Interface

Corman Lisp may be embedded in another application, and functions defined in lisp may be exported and called directly via another application. This is most-easily accomplished via COMPILE-DLL, which allows a Windows-standard DLL to be produced and linked to a client application. However, for more flexible and creative use of Corman Lisp within an application, the DirectCall interface may be used. The DirectCall interface is used internally by any DLL created using the Corman Lisp COMPILE-DLL command.

The DirectCall interface consists of several functions that are exported by the Corman Lisp kernel DLL CormanLispServer.dll. These are easily callable from C, C++ or other languages.

C/C++ API

Initialize

```
extern "C" long Initialize(
    const wchar_t* imageName,
    int clientType,
    HINSTANCE appInstance,
    HANDLE* thread,
    HWND mainWindow,
    TextOutputFuncType textOutputFunc);
```

imageName	The wide character file name (0-terminated) of the lisp image to load
clientType	0 = Windows application, 1 = console app, 2 = IDE
appInstance	HINSTANCE supplied to the application at startup, or to the DLL at load time
thread	Current thread handle. This should be a real handle, not the value returned by GetCurrentThread() (which is a pseudo-handle). You can use DuplicateHandle to obtain a real handle from the pseudo-handle returned by GetCurrentThread() if necessary.
mainWindow	The process main window handle. You may pass 0 (NULL) if unknown.
textOutputFunc	A function that can output text (generally, text written to *terminal-io*). The function should be defined like this: void OutputText(wchar_t* text, long numBytes);

BlessThread

```
extern "C" void BlessThread();
```

Causes the current thread (the thread calling this function) to be initialized as a lisp thread, to enable it to call lisp functions directly. This is necessary so the garbage collector knows what areas of the stack to scan during collection.

UnblessThread

```
extern "C" void UnblessThread();
```

This is the inverse operation to `BlessThread()`, and should be called when a thread is finished calling lisp functions.

GetCallbackFunctionPointer

```
extern "C" void* GetCallbackFunctionPointer(
    wchar_t* functionName,
    wchar_t* package);
```

`GetCallbackFunctionPointer()` is used to retrieve a pointer to a C function which is the exported entry to a lisp function.

functionName	Wide character string (0-terminated) of lisp callback function. This should have been defined in the current lisp image using `DEFUN-DIRECT-CALLBACK` or `DEFUN-DIRECT-C-CALLBACK`.
package	Wide character string (0-terminated) of the package name in which the specified direct callback function name resides.

Lisp API

DEFUN-DIRECT-CALLBACK

DEFUN-DIRECT-CALLBACK *name arglist &rest body*   *[macro]*

Package: `C-TYPES`

name is a symbol.

arglist is a list of the form:

```
        ((arg1-name  arg1-c-type)
         (arg2-name  arg2-c-type)
         etc.)
```

This macro defines a callback function which may be called from a client application via the DirectCall API. The function that is created has :pascal (__stdcall) linkage, which is the usual style for operating system callbacks in Windows. In other words, the callback function will remove its own parameters from the call stack.

This macro is similar in form and function to the `CT:DEFUN_CALLBACK` macro.

DEFUN-DIRECT-C-CALLBACK

DEFUN-DIRECT-C-CALLBACK *name arglist &rest body*   *[macro]*

Package: `C-TYPES`

This macro is like `DEFUN-DIRECT-CALLBACK`, except that the defined callback function has :c linkage, i.e. the caller is expecting to remove the passed parameters from the stack.

20 Threads

The Corman Lisp kernel has been designed to accommodate multiple threads of execution. Normally, when running Corman Lisp, at least two threads are executing in the current process. The first is the thread spawned by the application when it started. If you are running the Corman Lisp IDE, this will be the main user interface thread for the editor. If Corman Lisp was started by a different application, such as the `CLConsole.exe`, the console is running in this first thread. The second thread is known as the Lisp primary thread. It is the primary thread of execution of the Corman Lisp kernel. This thread is started when Corman Lisp is started, and only exits when Corman Lisp completely shuts down. While Corman Lisp is executing, other threads Lisp threads may be started. These each have their own stack.

The functions which support multiple threads are defined in the THREADS package, which has the nickname TH.

Special Variables and Threads

Because much of Common Lisp behavior is controlled by the binding of special variables at run time, each Lisp thread has its own special variable bindings. Corman Lisp has been designed such that the outermost special variable binding is shared between all Lisp threads, but when any special variables are rebound during execution, these new bindings may only be seen by functions executing in the thread which created the binding.

Example:

```
(defun foo () (let ((*print-base* 9)) (print 100)))
(th:create-thread #'foo)
```

In this example, a thread is created, which runs the function FOO. When it begins executing, it shares the binding of `*print-base*` with the primary thread. Once it rebinds the variable, however, that binding is only seen by functions executing in the new thread, not by functions executing in the primary thread. This is important because in this case it would interrupt any printing which was being done in the primary thread.

Although the Corman Lisp kernel supports multiple threads, many of the Corman Lisp library functions (including Common Lisp library function implementations) are not fully thread safe. You may use multiple threads for experimentation but be aware that you may run into problems at this time. This will be remedied in a future release.

To use any function in the THREADS package, you must first execute the form:

```
(require 'THREADS)
```

CREATE-THREAD

CREATE-THREAD *func &key (report-when-finished t)*   *[function]*

Package: THREADS

funcs should be a function to be executed in the context of a newly created thread.

If *report-when-finished* is true, then a message is sent to the Lisp Worksheet when the thread terminates. It will normally terminate when the function that was called in the new thread terminates. It may terminate early because of an unhandled error or exception.

Returns an integer representing the thread ID of the newly created thread.

EXIT-THREAD

EXIT-THREAD *condition*   *[function]*

Package: THREADS

Causes the calling thread to exit.

THREAD-HANDLE

THREAD-HANDLE *thread-id*   *[function]*

Package: THREADS

Returns the HANDLE associated with the passed thread id. The passed thread-id should be the thread id of a currently allocated lisp thread.

SUSPEND-THREAD

SUSPEND-THREAD *thread-id*   [*function*]

Package: THREADS

Suspends the lisp thread specified by the passed thread id.

RESUME-THREAD

RESUME-THREAD *thread-id*   [*function*]

Package: THREADS

Resumes the lisp thread specified by the passed thread id.

TERMINATE-THREAD

TERMINATE-THREAD *thread-id*   [*function*]

Package: THREADS

Terminates the lisp thread specified by the passed thread id. This is a brute-force termination, and is usually not the best way to end a thread. Better is to allow the thread to call EXIT-THREAD on its own.

CURRENT-THREAD-ID

CURRENT-THREAD-ID   [*variable*]

Package: CCL

This variable is always bound to the thread ID of the current thread.

CURRENT-THREAD-HANDLE

CURRENT-THREAD-HANDLE   [*variable*]

Package: CCL

This variable is always bound to the thread handle of the current thread.

CURRENT-PROCESS-ID

CURRENT-PROCESS-ID   [*variable*]

Package: CCL

This variable is always bound to the process ID of the current process.

CURRENT-PROCESS-HANDLE

CURRENT-PROCESS-HANDLE [*variable*]

Package: CCL

This variable is always bound to the process handle of the current process.

CRITICAL-SECTION

CRITICAL-SECTION [*class*]

Package: THREADS

Critical sections are CLOS classes which encapsulate a Win32
CRITICAL_SECTION object. These can be used for thread synchronization.
Consult a Win32 reference for details.

The two generic methods implemented for CRITICAL-SECTION instances are
ENTER and LEAVE. The accessor for the critical section itself is CS.

ALLOCATE-CRITICAL-SECTION

ALLOCATE-CRITICAL-SECTION [*function*]

Package: THREADS

Low-level function to allocate a Win32 CRITICAL_SECTION object. Note that
these should be deallocated when you are finished with them. However, if you
don't deallocate them the garbage collector will eventually do this via a
finalization function.

DEALLOCATE-CRITICAL-SECTION

DEALLOCATE-CRITICAL-SECTION *critical-section-handle* *[function]*

Package: THREADS

Low-level function to deallocate a Win32 CRITICAL_SECTION object.

ENTER-CRITICAL-SECTION

ENTER-CRITICAL-SECTION *critical-section-handle*   *[function]*

Package: THREADS

Low-level function to enter a protected code section using a Win32 CRITICAL_SECTION object. Note that you **must** call LEAVE-CRITICAL-SECTION for every call to ENTER-CRITICAL-SECTION. Because this is so important it is recommended you use UNWIND-PROTECT or the convenient WITH-SYNCHRONIZATION macro (defined below).

LEAVE-CRITICAL-SECTION

LEAVE-CRITICAL-SECTION *critical-section-handle*   *[function]*

Package: THREADS

Low-level function to leave a protected code section using a Win32 CRITICAL_SECTION object.

WITH-SYNCHRONIZATION

WITH-SYNCHRONIZATION *critical-section-handle . body [function]*

Package: THREADS

Example:

```
(with-synchronization (package-sync package)
       (#| do stuff to package |#))
```

Using this macro automatically calls ENTER-CRITICAL-SECTION before body, and ensures that LEAVE-CRITICAL-SECTION is called when the body finishes. If a non-local exit happens the critical section is still properly exited.

21 Lisp Data Structures

In Corman Lisp, all Lisp objects are tagged with information which allows their types to be determined at run time. Small datatypes, specifically those which fit in 29 bits or less, are stored as immediate values, with a type tag in the low order three bits. These include fixnums (integers in the range –268435455 to 268435455) and characters.

All other Lisp objects are stored on the heap, and are internally represented by a tagged pointer to a heap object. The low order three bits of this pointer are used to determine the type of data that is represented.

There are two types of heap objects, conses (two cells, with no header) and uvectors. These objects are interspersed on the heap. Uvectors have a block header in the first cell (with a tag of 110 in the lower three bits to indicate a special heap block header) which is followed by an odd number of cells (to make the total block an even number of cells). Thus, all heap blocks are a multiple of 8 bytes in size. Because of this, the lower 3 bits of any pointer to a heap block would normally be 000, and the type tag can be stored in these bits. For cons cells, the type tag is 100, and for uvectors it is 101.

There are some primitive functions which can be used to access any type of Lisp heap object in a very efficient manner. These are all inlined by the compiler automatically. These are dangerous functions to use, and are only documented here for informational purposes or to use at your own risk. You should know what you are doing if you use them. They are used throughout the Corman Lisp source code.

Lisp Object Functions

UREF

    UREF *uvector fixnum*   *[function]*

    Package: `CCL`

    uvector is any Lisp heap object other than a cons cell.

    fixnum is an integer in the range 1 .. size of uvector.

Returns the contents of the requested cell, with 0 being the header cell. However, you should not access the header cell directly, as the contents of that slot are not a valid Lisp object. The special tag used for block headers is not a legal object, and will cause most functions to misbehave if they see one. The fact that it is not a legal Lisp object is useful for scanning the heap, and for other internal uses. All other cells either contain a valid Lisp object, or contain the special value `UNINITIALIZED`. This value is also dangerous. It is best to use `UNINITIALIZED-OBJECT-P` to test the contents of a slot before accessing it. Because `UREF` is a low level primitive, little type checking is done when you use it.

`SETF` may be used with `UREF` to store the contents of a slot.

UVECTORP

UVECTORP *object*   *[function]*

Package: CCL

object   is any Lisp object.

Returns T if the passed object is a uvector, `NIL` otherwise.

Type Tags

The contents of all Lisp cells (which include stack-allocated variables, heap objects and static objects) can be divided into the following groups, based on their tags (low three bits).

Tag	Type
000	Fixnum (29-bit integer)
001	Character, other small immediate objects
010	Forwarding tag. Used by the garbage collector only.
011	Immediate short float
100	Cons reference
101	Uvector reference
110	Uvector header tag. Only found in the first cell of a uvector.
111	Immediate short float

Immediate Data Types

The following types are stored directly in a cell.

Fixnum

Bits	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3	2 1 0
Contents	29-bit signed fixnum	0 0 0

Character

Bits	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4 3	2 1 0
Contents	Unused	16-bit character (normally ASCII)	0 0 0 0 0	0 0 1

Short Float

Bits	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2	1 0
Contents	30-bit short float	1 1

Heap Object References

The following types are stored as a reference to a heap object. For a cons cell, the reference has this format:

Cons Reference

Bits	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3	2 1 0
Contents	Upper 29 bits of a 32-bit pointer to a cons heap object	1 0 0

For all other heap objects, the reference is in this format:

Uvector Reference

Bits	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3	2 1 0
Contents	Upper 29 bits of a 32-bit pointer to a uvector heap object	1 0 1

Extended Data Types

Here is the complete list of heap objects supported in Corman Lisp.

Note that the block length is a 24-bit integer representing the number of cells /2 which comprise the block. All cell contents are tagged (in the lower 3 bits) unless otherwise noted.

Cons

Cell	Contents		
0	32-bit Lisp object		
1	32-bit Lisp object		

Lisp Function (closure)

Cell	Contents		
0	Uvector length = 2	0 0 0 0 0	1 1 0
1	Environment		
2	Compiled-code block		
3	Unused		

Kernel Function

Cell	Contents		
0	Uvector length = 2	0 0 0 0 1	1 1 0
1	Environment (NIL)		
2	Pointer to foreign function (non-tagged)		
3	Unused		

Structure

Cell	Contents		
0	Uvector length	0 0 0 1 0	1 1 0
1	Structure definition vector		
2	Slot 1		
3	Slot 2		
4	Etc.		

Array (adjustable)

Cell	Contents		
0	Uvector length	0 0 0 1 1	1 1 0
1	Vector (simple, float, char, etc.)		
2	Fill pointer		
3	Offset (displaced arrays)		
4	Number of dimensions		
5	Dimension 0 length		
6	Dimension 1 length		
7	Dimension 2 length		
8	Etc. (only as many dimensions as are needed are allocated)		

Symbol

Cell	Contents		
0	Uvector length = 5	0 0 1 0 0	1 1 0
1	Print value		
2	Value		
3	Package		
4	Property list		
5	Flags		
6	Function type		
7	Function		
8	Jump table entry (if non-zero)		
9	Variable table entry (if non-zero)		

Stream

Cell	Contents		
0	Uvector length = 11	0 0 1 0 1	1 1 0
1	Name		
2	Underflow function		
3	Position		
4	Column		
5	Input buffer		
6	Input buffer length		
7	Input buffer position		
8	Input buffer size		
9	File handle or string		
10	Common Lisp subclass		
11	Binary (boolean)		
12	Open (boolean)		
13	Direction (:input, :output, :bidirectional or :nil)		
14	Interactive (boolean)		
15	Element type		
16	Associated streams		
17	Overflow function		
18	Output buffer		
19	Output buffer length		
20	Output buffer position		
21	Line number		

Double Float

Cell	Contents		
0	Uvector length = 2	0 0 1 1 0	1 1 0
1	Unused		
2	Float (upper 32 bits, non-tagged)		
3	Float (lower 32 bits, non-tagged)		

Package

Cell	Contents		
0	Uvector length = 5	0 0 1 1 1	1 1 0
1	Name		
2	Nicknames		
3	Use-packages		
4	Used-by packages		
5	Shadowing symbols		
6	Current capacity		
7	Number of symbols currently stored		
8	Table (capacity * 3 cells)		
9	Unused		

Hash-table

Cell	Contents		
0	Uvector length = 3	0 1 0 0 0	1 1 0
1	Current capacity		
2	Number of objects currently stored		
3	Test function		
4	Vector (capacity * 2 cells)		
5	Unused		

Foreign Pointer

Cell	Contents		
0	Uvector length = 1	0 1 0 0 1	1 1 0
1	Foreign pointer (untagged)		

Compiled-code Block

Cell	Contents		
0	Uvector length	0 1 0 1 0	1 1 0
1	Magic number identifier: 0xDEADBEEF		
2	References vector		
3	Properties list		
4	Code bytes 0 - 3		
5	Code bytes 4 - 7		
6	Etc.		

Readtable

Cell	Contents		
0	Uvector length = 3	0 1 0 1 1	1 1 0
1	Read level (integer)		
2	Backquote processing (boolean)		
3	Table (256 * 2 cells)		
4	Case		
5	Unused		

Complex Number

Cell	Contents		
0	Uvector length = 2	0 1 1 0 0	1 1 0
1	Real component		
2	Imaginary component		
3	Unused		

Ratio

Cell	Contents		
0	Uvector length = 2	0 1 1 0 1	1 1 0
1	Numerator		
2	Denominator		
3	Unused		

Bignum

Cell	Contents		
0	Uvector length	0 1 1 1 0	1 1 0
1	Number of 32-bit cells * 2 + sign		
2	Cell 0		
3	Cell 1		
4	Etc.		

Foreign Heap Pointer

Cell	Contents		
0	Uvector length = 2	0 1 1 1 1	1 1 0
1	Foreign heap ptr (non-tagged)		
2	Tagged size of heap block (in bytes)		
3	System tag (normally = 0)		

Weak Pointer

Cell	Contents		
0	Uvector length = 1	1 0 0 0 0	1 1 0
1	Weak reference to heap object		

Simple Vector

Cell	Contents		
0	Uvector length	1 0 0 0 1	1 1 0
1	Vector length		
2	Cell 0		
3	Cell 1		
4	Etc.		

Simple Character Vector

Cell	Contents		
0	Uvector length	1 0 0 1 0	1 1 0
1	Vector length		
2	Chars 0 – 1 (non-tagged)		
3	Chars 2 – 3 (non-tagged)		
4	Etc.		

Simple Byte Vector

Cell	Contents		
0	Uvector length	1 0 0 1 1	1 1 0
1	Vector length		
2	Bytes 0 – 3 (non-tagged)		
3	Bytes 4 – 7 (non-tagged)		
4	Etc.		

Simple Short Vector

Cell	Contents		
0	Uvector length	1 0 1 0 0	1 1 0
1	Vector length		
2	Shorts 0 – 1 (non-tagged)		
3	Shorts 2 – 3 (non-tagged)		
4	Etc.		

Simple Double Float Vector

Cell	Contents		
0	Uvector length	1 0 1 0 1	1 1 0
1	Vector length		
2	Float 0 (low 32-bits) (non-tagged)		
3	Float 0 (high 32-bits) (non-tagged)		
4	Float 1 (low 32-bits) (non-tagged)		
5	Float 1 (high 32-bits) (non-tagged)		
6	Etc.		

Simple Bit Vector

Cell	Contents		
0	Uvector length	1 0 1 1 0	1 1 0
1	Vector length		
2	Bits 0 - 3 (one per byte) (non-tagged)		
3	Bits 4 - 7 (one per byte) (non-tagged)		
4	Etc.		

Simple Single Float Vector

Cell	Contents		
0	Uvector length	1 0 1 1 1	1 1 0
1	Vector length		
2	Single float 0		
3	Single float 1		
4	Etc.		

Single Float

Cell	Contents		
0	Uvector length = 1	1 1 0 0 0	1 1 0
1	Single float		

CLOS Instance

Cell	Contents		
0	Uvector length = 2	1 1 0 0 1	1 1 0
1	Class		
2	Slots		
3	Signature		

22 Corman Lisp Extensions

Here are some non-standard functions and variables which are included in Corman Lisp and which you may find useful.

Non-standard Functions and Variables

TOP-LEVEL

TOP-LEVEL   [*variable*]

This should normally be bound to the top-level read-eval-print loop.

SAVE-IMAGE

SAVE-IMAGE *pathname*   [*function*]

The SAVE-IMAGE command (in the Corman Lisp package) is used to dump the Lisp heap to a file for later reloading. Invoking this function causes all ephemeral heaps to be flushed into the primary heap (GC 3) and then the entire heap is written to the file specified by the passed path name. You must be careful how you use this function. It handles Lisp objects very well, and is very fast. However, non-Lisp objects, such as Windows resources (handles, etc.) will not be saved and restored properly without specific intervention by code that you need to write.

Foreign heap objects will be properly saved and restored, with the exception that they may load at a different address later. If one foreign heap object has a pointer to another foreign heap object (or back to itself) this could be a problem. Foreign function definitions and callbacks, and COM interfaces (which allocate foreign heap objects internally) are handled correctly.

If the *pathname* specifies a file with an extension .EXE, then the saved image is appended to the .EXE file as a .lisp segment. Any contents of the .EXE file are preserved. Note that if a .lisp segment already exists, another one will be created. This is probably not what you want. In future versions, the existing .lisp segment will be overwritten.

This function will create an image in the **output directory** if absolute image pathname was not given *(See Section 6, Subsection "User Specific Files and Directories" for details)*.

LOAD-IMAGE

LOAD-IMAGE *pathname* [*function*]

This function (in the Corman Lisp package) is passed the name of a file created with the SAVE-IMAGE function. The saved Lisp heap is restored. This is a very fast operation, almost nothing needs to be done except read the file into memory. This function is called internally when Corman Lisp starts executing. By default the file CormanLisp.img is loaded. However, it is possible to specify a different image file as a command line parameter when invoking CormanLisp.exe or CLConsole.exe.

If the *pathname* specifies a file with an extension .EXE, then the loaded image is read from a .lisp segment of the .EXE file. Any other contents of the .EXE file are ignored. Note that if more than one .lisp segment exists in the file, only the first one will be read.

This function will will look for an image in the **output directory** if absolute image pathname was not given *(See Section 6, Subsection "User Specific Files and Directories" for details)* or in the **installation directory** (as the last resort).

LOAD-DEFAULT-IMAGE

LOAD-DEFAULT-IMAGE [*function*]

Package: CCL

This operator takes no arguments, and loads the CormanLisp.img heap file in the root install directory. The default Lisp heap is restored.

CURRENT-DIRECTORY

CURRENT-DIRECTORY [*function*]

This function (in the Corman Lisp package) returns the current directory.

SETF CURRENT-DIRECTORY

(SETF (CURRENT-DIRECTORY) *pathname)* [*function*]

This function (in the Corman Lisp package) sets the current directory.

FUNCTION-ENVIRONMENT

FUNCTION-ENVIRONMENT *function* [*function*]

Package: CCL

Returns the lexical environment of any function. Because the Corman Lisp compiler optimizer automatically turns any lexical variables it can into stack allocated variables, only actual captured bindings will be contained in the returned environment i.e. only those lexical variable bindings which persist after the function that created the closure has exited.

Example:

```
(let ((count 0))
    (defun get-count () count)
    (defun set-count (x)(setf count x)))

(function-environment #'get-count)

#S( COMMON-LISP::ENVIRONMENT :SLOT1 (0) :SLOT2 #< Uninitialized > )
```

You should not attempt to manipulate the returned environment. However, if
you really feel you need to, you may use:

```
    (UREF environment 2) -> slot 1 binding
    (UREF environment 3) -> slot 2 binding
    etc.
```

In each case, UREF returns the binding, which is a cons cell whose CAR
represents the current value of the binding.

CORMANLISP-DIRECTORY

CORMANLISP-DIRECTORY   [*variable*]

Package: CCL

At run time this variable is bound to the directory the main application was
launched from.

CORMANLISP-SERVER-DIRECTORY

CORMANLISP-SERVER-DIRECTORY   [*variable*]

Package: CCL

At run time this variable is bound to the directory that CormanLispServer.dll
was launched from.

FIND-IN-FILES

FIND-IN-FILES *string path* &key **(**ignore-case t**))**   [*function*]

Package: CCL

Searches *path,* which may contain wildcards, for the passed string.

Example:

```
(find-in-files "corman" "sys/*lisp")
```

MACROEXPAND-ALL

MACROEXPAND-ALL *expression*   [*function*]

Package: CCL

Like MACROEXPAND, but recursively descends the expression and expands
all macros in the expression. This is the same function used by the compiler
whenever it compiles a function or expression. It does not descend inside
embedded LAMBDA expressions, however.

Example:

```
(ccl:macroexpand-all '(defun foo (x) x))
```

LISP-SHUTDOWN

LISP-SHUTDOWN *message*   [*function*]

Package: CCL

This function shuts down the Lisp application cleanly, which may be the Corman
Lisp IDE, a console application, or any other Corman Lisp hosting application
(including any applications created via SAVE-APPLICATION). It does this by
both quitting Corman Lisp threads, and then instructing the hosting application
(via COM) to shut itself down. This is cleaner and more reliable than calling
EXIT-PROCESS.

The passed message argument should be a string which will be displayed to the
user when the process exits.

USE-EXTERNAL-BROWSER

USE-EXTERNAL-BROWSER   [*variable*]

Package: CCL

Use default system WEB-browser for documentation lookups.

23 Auto-Update

An auto-update feature, enabled by default in the Corman Lisp IDE, will keep your Corman Lisp installation up to date with patches, documentation updates, etc. Each time the IDE starts it will check the Corman Lisp web site (www.cormanlisp.com) for new patch files. The patch level of these files will be compared against the patch level of your installed version, and if the patches are newer, you will be prompted to upgrade to the newer patch level. If you choose to upgrade, the new patches will be loaded and installed automatically, and a new CormanLisp.img file will be generated. Previous files will be backed up, and may be used later if for some reason you choose to roll back (undo) a patch.

The Auto-Update facility may be used from the Corman Lisp console as well, but does not run automatically (by default). You may make changes in the init.lisp file to either disable Auto-Update in the IDE, or enable it in the console, as desired.

While the basic feature is automatic, the following functions exported by the CormanLisp (CCL) package may be useful to you.

CORMANLISP-PATCH-LEVEL

CORMANLISP-PATCH-LEVEL   [*variable*]

Package: CCL

At run time this variable is bound to current patch level (an integer). Patch level 0 means no patches have been applied to the release version.

AUTO-UPDATE

AUTO-UPDATE   [*function*]

Package: CCL

Invokes the Auto-Update feature.

PATCH-ROLLB ACK

PATCH-ROLLBACK *level*   [*function*]

Package: CCL

The argument should be an integer between 0 and the current patch level. This function causes any patches newer than the specified level to be rolled back (undone).

AUTO-UPDATE-ENABLED

AUTO-UPDATE-ENABLED   [*variable*]

Package: CCL

This is true by default. If you set it to NIL, (in the `init.lisp` file, for example), then the Auto-Update feature is disabled.

PATCH-ROOT-DIRECTORY

PATCH-ROOT-DIRECTORY   [*variable*]

Package: CCL

This variable is set to the directory path on the www.cormanlisp.com server which contains patches. You should not need to set or change this variable, and it is documented here for informative purposes only.

PATCH-SERVER

PATCH-SERVER   [*variable*]

Package: CCL

This variable is set to the server www.cormanlisp.com. You should not need to set or change this variable, and it is documented here for informative purposes only.

AUTO-UPDATE-LEVEL

AUTO-UPDATE-LEVEL   [*variable*]

Package: CCL

This is used by the Auto-Update facility to allow updates to the Auto-Update feature itself. (This was necessary at one point during 3.0 development). You should not need to set or change this variable, and it is documented here for informative purposes only.

COMPILE-CORMANLISP-IMAGE

COMPILE-CORMANLISP-IMAGE [*function*]

Package: `CCL`

This operator takes no arguments, and recompiles the `CormanLisp.img` file in the root install directory. Normally it is automatically called by Auto-Update, but you may also call it explicitly if you have modified the kernel or any files in the `SYS/` directory. After invoking it, a console process is started which executes the operation. When the spawned console finishes, you should restart Corman Lisp (either the IDE or the console app, depending which you are running) and you will now have the updated `CormanLisp.img` file.

24 Memory Report

Corman Lisp contains a system dump feature called MEMORY-REPORT which causes a dump file of all threads, stacks, registers and heaps to be created. This can be useful for tracking down system problems, FFI issues, threading issues, etc. which are beyond the capabilities of the built-in Lisp debug facility. The dump file is created in the Corman Lisp. The dump file is located in the Corman Lisp root install directory, and is called `CormanLisp_dump.log`. When this function is invoked, any previous dump file is automatically overwritten.

Various system errors will automatically generate this dump file when a serious error is detected. In some cases the system will attempt to continue execution from a known restart, such as the TOP-LEVEL function. In other cases, such as when the system was in the process of garbage collection when the error was detected, recovery is not possible and the lisp system shuts itself down. If you are running in the IDE you may still be able to continue to edit files and save open files, but Lisp code will not execute.

When the memory report function is called, a new thread is started (not a Lisp thread) and all Lisp threads are halted. The new thread scans all Lisp threads and reports their register contents as well as a dump of the stack of each thread. The stack dump attempts to locate stack frame links and to determine which function was executing at each level. This is not always possible and so you may not always be able to rely on the function names being reported. Various annotations are placed next to contents of the stack to enable diagnosing of problems.

Finally, the contents of all three Lisp heaps (the two ephemeral heaps and the primary heap) are dumped. Each object on the heap is either a CONS (2 cells, 8 bytes, with no header) or a UVECTOR—an arbitrary number of cells with the first cell containing a uvector header (see the chapter on Run-Time Architecture for details).

This dump file is usually quite large (20 megs to 40 megs is common). If you need to view it, make sure you use an appropriate text editor which can handle large files. For example, don't try to use Microsoft Word. Notepad works OK. I like to use TextPad (www.textpad.com).

MEMORY-REPORT

MEMORY-REPORT [*function*]

Package: CCL

This function takes no arguments, and creates the dump file
CormanLisp_dump.log. It will overwrite any existing file by that name.

25 Microsoft .NET Integration

Corman Lisp includes the RDNZL package from Edi Weitz. This package enables Lisp code to call methods on .NET objects, as well as read and set properties and perform other .NET actions. The syntax is very nice and some sample applications are provided.

Dr. Weitz has been kind enough to allow us to distribute the RDNZL package with Corman Lisp, while retaining the following copyright on all files:

Copyright (c) 2004-2006, Dr. Edmund Weitz. All rights reserved.

To enable .NET integration, use the following command:

```
(require :RDNZL)
```

This loads a file modules/rdnzl.lisp, which is provided by Corman Lisp. This in turn invokes the RDNZL loader provided by the release package of RDNZL. We have set this up so that you may upgrade to a different version of RDNZL without having to change any of your source files.

The first time you load RDNZL, it will attempt to compile all the files using COMPILE-FILE. This will take a few seconds. Each time that you load RDNZL on subsequent sessions, the compiled FASL files will be reloaded and the load should be much faster.

RDNZL-VERSION

RDNZL-VERSION   [*variable*]

Package: CCL

This variable contains the currently installed (or preferred) version of RDNZL. If you wish to update the RDNZL package, you should install it in the libraries/rdnzl/ path under the root Corman Lisp directory (ccl:*cormanlisp-directory*). You should then update the value of this variable in the file modules/rdnzl.lisp.

RDNZL-PATH

RDNZL-PATH   [*variable*]

Package: `CCL`

This variable contains the path to the installed RDNZL package. It is set by the system, based on the version string `(*RDNZL-VERSION*)`.

If you wish to learn more about .NET integration via RDNZL, please see the file `doc/index.html` under the RDNZL package directory. Once you have loaded RDNZL, you can view the help file in the Corman Lisp IDE with the following command:

```
(ccl:display-url
      (concatenate 'string ccl::*rdnzl-path* "doc/index.html"))
```

This file will also be available from the **Help** menu in the Corman Lisp IDE.

26 Run-time Architecture

This section is designed to give you information that you need if you want to go below the surface of Corman Lisp, such as writing assembly code, special code generators, or custom interface code.

Any code which is executed, whether written in Lisp as part of the system or by a user, or kernel code written in C, or assembler code, exists as machine code at the time it executes. The Corman Lisp system, at run time, integrates code from all these sources with code that you write. They all have to work harmoniously together, or things would get ugly quickly.

Here are a few of the specific issues that can be problems, and which any generated code needs to account for:

- Lisp functions generally can take any number of parameters, and return any number of values. Functions written in C tend to take a fixed number of parameters, and return a single value. In assembler, anything is possible.

- C functions like to have their parameters pushed on the stack from right to left (last parameter first), but Common Lisp code works better if parameters are pushed on the stack from left to right.

- All Lisp heap objects can move in memory at any time, because of garbage collection. A memory address is not a good way to keep track of a particular lisp object. It is particularly bad to have a pointer which points into the middle of a lisp object. In C, on the other hand, pointers into the middle of arrays and structures are common, and in fact typically get generated by compilers.

- The presence of foreign code (either called by Lisp code or calling Lisp code) violates many of Lisp's run-time rules. To handle this, a certain amount of run-time record-keeping must be maintained.

- The presence of multiple Lisp threads, possibly running simultaneously on multiple processors, means that Lisp code must be very careful not to allow two threads to deadlock or to simultaneously try to write to the same heap object.

These are just a few of the issues that make generating Lisp code a challenge. In short the main areas we need to consider are: the lisp function runtime architecture, foreign code, garbage collection, and multiple threads.

We will start by looking at how a particular lisp function executes. To do this we need to delve into Intel x86 assembler. This section assumes the reader knows a bit about assembler. If not, you probably need to do a little research on your own to fully understand this section.

An important concept to mention here is tagging. All lisp objects are tagged with 2 or 3 bits of type information in the low bits of the 32-bit word. For example, fixnums have the low three bits 000. When describing the runtime architecture, we need to distinguish when a pointer or integer is tagged or untagged. Typically they are always tagged, except in certain circumstances.

Register Usage Conventions

The processor has the following 32-bit registers (we will ignore floating point registers here):

EIP	Instruction Pointer	Always contains the address of the instruction being executed (or about to be executed).
EBP	Base Pointer	Always contains the address of the current stack frame. Therefore it points into the stack of the currently executing thread. Negative offsets from EBP are usually local variables, and positive offsets are passed parameters.
ESI	QV Pointer	Always contains the address of the current thread's QV vector. This is a vector containing kernel objects, jump table entries, and dynamic variable bindings. It is not part of the Lisp heap. QV[0] is always the symbol NIL, therefore MOV eax, [esi] moves NIL into the eax register.
EDI	Function Environment Pointer	On entry to a lisp function, this register contains the address of the function's environment. This environment contains lexical bindings which were captured when the function was compiled. If there were no lexical bindings (from outer functions) which were captured at compile time, the environment is empty, and EDI will point to NIL. During execution of the function, EDI may be reused for other purposes.

EAX	Function Return	EAX is a general purpose register, used for many temporary calculations. It is specifically used to return a value from a function. If a function returns a single value, EAX always holds that value. If the function returns more than one value, EAX holds the first value, and all the values are kept in a list at address {ESI + 8]. If no values are returned, EAX will contain NIL.
EBX	Parameter Block	On entry to any function which takes an arbitrary number of parameters, EBX is computed and initialized to point to the first (leftmost) parameter. Note that this is not at a fixed offset from the base pointer (EBP) but has to be computed based on the number of parameters actually passed. If a function takes a fixed number of parameters, or when EBX is no longer needed, EBX is used as a general purpose register.
ECX	Parameter/ Return Value count	On entry to any lisp function, ECX contains the untagged number of parameters passed to the function. When a function returns, ECX contains the untagged number of values being returned. When not calling or returning from a function, ECX is a general purpose register.
EDX	General Purpose	This register has no special usage, and is generally used, along with EAX, for temporary calculations.

Tagging

We noted that lisp data is tagged in the low bits with some type information. The details of tag bits are discussed in a previous section of this document (see *Lisp Data Structures*). We need to expand on the notion of tagging a bit more here, to fully explain its purpose and when untagged values are allowed.

The primary purpose for tagging is to allow the garbage collector to distinguish between pointers to heap objects, which it needs to be able to relocate, and arbitrary data like numbers or characters. When the garbage collector moves an object, it must update the pointer in memory to point to the new address. It is essential that it always can correctly distinguish between heap pointers and data. If it mistook an integer for a heap address, it could corrupt the heap (by moving only part of a block of data) as well as corrupt the integer (by changing it into a pointer to the new heap address).

Heap allocations happen all the time during lisp code execution. It is nearly impossible to prevent them, and they are fast enough you shouldn't need to (on par with stack frame allocation). But since any heap allocation can trigger garbage collection, and this can even happen from another lisp thread, we need to keep all lisp data tagged at all times. This includes not only all data in the heap, but also all data on the stack and in registers.

Having said that, we will more precisely specify what it means to be *safe*. We will say data is safe, in relation to memory management, if it is either tagged or by some other means will not be mistaken for the wrong type by the garbage collector. By the most convenient definition, we will say something is safe if it is data (not a pointer) and cannot be mistaken for a pointer, or if it is a pointer and cannot be mistaken for data. Thus we will consider, for this discussion, that the number of passed parameters to a function is correctly tagged, even though it does not contain 000 in its low three bits (it is an untagged integer). This is OK because the number of parameters cannot exceed 65536, and all heap addresses are guaranteed to be above 65536. This bit pattern will never confuse the garbage collector. In general, bit patterns below 65536 are always safe, and need not be tagged. Note that they must be stored in a full 32-bit register, however. If we were to store one of these values in the low 16-bits of a register, and another small integer in the upper 16-bits, we could inadvertently create what looks like a heap address.

The garbage collector assumes something is a pointer if it points into the range of the heap i.e. it contains an address which is within the bounds of some section of the heap, and if it contains a tag of 100 or 101 in the lower 3 bits of the address. These represent CONS and UVECTOR heap objects, respectively. A CONS is an 8-byte heap object, with no header, and a UVECTOR is an arbitrary-sized heap object with a 4-byte header. All UVECTORs are multiples of 8 bytes in size, which means all heap objects begin on 8-byte boundaries.

Any bit pattern in a register which contains other than 100 or 101 in its low three bits, or does not point within an area of the lisp heap, is assumed by the garbage collector to be data. With a single exception:

In x86 assembly language, code instructions vary from 1-15+ bytes, and are byte-aligned, meaning no special alignment. The EIP register is always known to contain a pointer to code, and the garbage collector determines whether this is a pointer to a lisp code block (which is a lisp heap block) simply by looking at its address and seeing if it is within a heap section. The logic to handle this pointer is a little trickier, because it points into the middle of a heap block rather than the beginning. Also, a return address is stored on the stack along with every stack frame, and these return addresses are like the EIP contents in terms of how they need to be handled. The garbage collector must be able to precisely identify all the return addresses on the stack, and move the code they point to and update them correctly.

Some lisp heap objects may contain binary data which is not tagged. These objects include single-floats, bignums, byte arrays, strings, bit arrays, etc. In all cases these are stored in uvectors. In the uvector header, 5 bits are used to indicate the uvector type (which includes all the types just mentioned). The garbage collector, as it scans the heap, uses this type information to determine which bits to skip during collection. This prevents it from mistaking arbitrary bit patterns for heap pointers.

Atomic Operations

Sometimes it is necessary for executing code to temporarily hold a bit pattern in a register which is unsafe i.e. could be mistaken for the wrong type by the garbage collector. Since the collector can normally run between any arbitrary pair of instructions, there needs to be a mechanism to prevent the collector from interrupting a calculation when the some unsafe data is present. This is managed with a pair of instructions called:

```
BEGIN-ATOMIC
END-ATOMIC
```

When the collector runs, it suspends each lisp thread (except itself, which is known to be in a safe place). It examines each thread, and if any thread is executing an atomic block, it resumes that thread for a few milliseconds, and then suspends it again. It continues this until the thread is in a non-atomic instruction region. The term atomic is used here to mean that the atomic block cannot be interrupted by the garbage collector. It does not mean that it can't be interrupted by another thread. It can. You have to use synchronization objects to deal with thread interruption issues (which is a different topic).

In terms of implementation, a flag in the control register (the direction flag) is borrowed for use here. If the direction flag is set (STD) then the thread is assumed to be in an atomic block. If it is cleared (CLD) then it is in its normal, non-atomic state. Of course the direction flag could be used for its intended purpose, such as setting the direction of repeated MOV operations. We get around this by the following:

Corman Lisp code never uses the direction flag for this purpose. Newer processors don't really get a performance benefit from using the instructions which pay attention to the direction flag, so we just avoid it in lisp code and kernel code.

Foreign code could still use it, so the logic the garbage collector uses to determine if an operation is atomic is that the executing code must be foreign, or else the direction flag clear, otherwise it assumes the thread is in an atomic block. The way it determines whether foreign code is executing will be discussed later.

Atomic blocks need to be used very judiciously. Generally, they should never span more than a few instructions, and another function should never be called within an atomic block. As a special case, the garbage collector considers entering a function and exiting a function to be atomic during the time that the stack frame is being created and freed. The direction flag is not set here—rather the collector examines the instructions and makes a determination about whether the thread is in either of these states.

A simple compiled function:

```
;;;;
;;;;Common Lisp IDENTITY function.
;;;;
(defasm identity (obj)
     {
       push    ebp                       ;; link new frame
       mov     ebp, esp
       cmp     ecx, 1                    ;; check number of
arguments
       je      short :next
       callp   _wrong-number-of-args-error
    :next
       mov     eax, [ebp + ARGS_OFFSET]  ;; return argument in eax
       mov     ecx, 1                    ;; returning a single value
       pop     ebp                       ;; unlink stack frame
       ret
    })
```

This is a very simple function which demonstrates a few important points. The first two instructions link a new stack frame. These must always be called, in precisely this way, at the start of any function. They cannot be optimized away, even if you are writing in assembler, because the garbage collector and lisp debugger rely on an intact linked list of stack frames. Of course foreign code may not follow this rule—there is nothing we can do about that. This is handled by marking areas of the stack that are used by foreign code, and skipping over them.

The next two instructions are simply to make sure that one and only one argument has been passed. These two instructions could be optimized away for performance reasons, with less safety. The compiler does just that if you turn up optimize speed and turn down optimize safety settings.

The next instruction fetches the single argument from the parameter block. The rightmost parameter (in this case the only one) is always located 8 bytes above EBP (the constant ARGS_OFFSET is used to make this clear) with any other arguments continuing at higher addresses. The first, or leftmost, parameters will be at the highest address.

As an aside: lisp code pushes parameters from left to right because it is required to evaluate them in that order. As they are evaluated, the results must be stored somewhere, and the stack is where they need to be eventually, so this makes the most sense. You might think they could as easily be stored in the opposite order on the stack, but that's not true. If we use a single instruction to allocate a block on the stack for parameters, then all the bytes in that block are uninitialized until we are done evaluating and filling in the parameter slots. If garbage collection occurs here,

the uninitialized slots will cause problems. Therefore, we need to either initialize all the slots immediately, which has a performance cost, or just push them on the stack as we evaluate them, left to right.

After storing the value to be returned in EAX, we set ECX to the number of values being returned. In this case this one instruction is unnecessary, because ECX is guaranteed to already contain 1 at this point, so that could be optimized away. Finally the last two instructions unlink the stack frame and return. These last two instructions are always required, exactly as presented here. Note that ESP must be equal to EBP prior to executing these last two instructions. If it wasn't you might see

```
MOV ESP, EBP
```

prior to the last two instructions.

Other things to note:

CALLP is a lisp assembler macro which translates into an indirect JSR instruction via the jump table located in the QV vector. You should always use CALLP and CALLF when calling functions from assembler code. Normally you will use CALLF, which is the same, but loads the EDI register with the function environment prior to the JSR. As an optimization, if you know the environment is empty and unused, you can use CALLP which skips that instruction.

Also note: _wrong-number-of-args-error never returns so we need not be concerned about what happens after we call it. It ends up calling ERROR and returning via a restart. Also note, you should use keywords as jmp targets. Using the *short* qualifier in any jump instruction means the target will be no more than 128 bytes in either direction (actually –128 to +127 bytes). In this case several bytes are saved in the code.

Credits

Corman Lisp Credits (updated Dec. 30, 2018)

Release 3.1 includes new contributions from the following people (in alphabetical order):

> Artem Boldariev (arbv@Github)
>
> Chun Tian (binghe@Github)
>
> Luis Cervantes (Luis-Cervantes@Github)
>
> Max Chumak
>
> Roger Corman (rgcorman@Github)
>
> Zachary Beane (xach@Github)

Release 3.0 includes new contributed work and assistance by a number of people, including:

> Edi Weitz
>
> Espen Wiborg
>
> Karsten Poeck

The Corman Lisp compiler, run-time system, IDE and most associated components and have been developed by Roger Corman. Several of the included components had other authors, however.

Frances Corman is now a part of the Corman Lisp team, and has revised the manual for this release. Ethan Stone, our senior software engineer, has made significant contributions to the 3.0 release including much of the new GUI functionality and the Othello sample game.

Edi Weitz has developed the RDNZL package for Common Lisp .NET integration, and has given us permission to distribute it with Corman Lisp (see the chapter on Microsoft .NET Integration).

Chris Double has a great web site at:

> http://www.double.co.nz/cl/index.htm

which contains all kinds of Corman Lisp add-on features and programs (which he has personally developed). He continues to provide excellent support to others, and

moderates the Corman Lisp message board. Many of Chris' works are included, with his permission, with the Corman Lisp product.

Vassili Bykov has provided the sophisticated C-header parser for loading foreign interface definitions. This is extremely nice and useful, and is included in full with the Corman Lisp package. He has also contributed translations of several Windows header files, as well as some Common Lisp library functions.

Reini Urban has done a lot of work to help make Corman Lisp function as an extension under Autocad. His modifications are included in this release.

The CLOS implementation is derived from the work of Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow and is very well documented in their book *Art of the Metaobject Protocol.* The code is Copyright (c) 1990, 1991 Xerox Corporation.

The random number facility is from the CMU Common Lisp project (CMU Lisp) and was originally written by David Adam.

Rob MacLachlan also contributed.

The LOOP facility is from MIT, and lists Glenn Burke as one of the authors. See the file LOOP.LISP for more details.

Kent Pitman has developed the wonderful HyperSpec, which can be browsed from within Corman Lisp. This is an incredible document that covers not only ANSI Common Lisp, but much of the language design process as well. Thanks for your efforts, Kent. In general, thanks to all the members of the ANSI Common Lisp standard committee X3J13 for standardizing such a powerful language.

Thanks to Guy Steele Jr., whose books *Common Lisp: The Language*, both editions, are a constant source of assistance and amusement (in the most positive sense). Corman Lisp also includes the optimized backquote facility from the second edition, as well as some other functions from the book.

Rainer Joswig's e-mail discussions with me early in the Corman Lisp development inspired several of the features of Corman Lisp, including the support for OS native threads.

Peter Norvig's book *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp* taught me a lot about the language and his many sample programs were useful in debugging the Corman Lisp compiler. The source to his Eliza program is included as an example with this release.

Paul Graham has been an inspiration as well, through his excellent books *ANSI Common Lisp* and *On Lisp.*

Source examples are included from numerous people, including Larry Hunter and
Vassili Bykov.

People who have been influential in my Lisp understanding over the years include:

    Arthur Hills

    Jim Bisso

    David Betz

    Henry Baker

    ... and many others.

Thanks also to all the licensed users of Corman Lisp and PowerLisp, who helped to support me over the years with advice as well as financial support.

Thanks to my wife Frances, children Amy and Emmett, and grandson Cyrus, for helping me and putting up with me when I have been unavailable or in need of sleep.

-Roger Corman   September 8, 2006

roger@corman.net