GenDL Unified Documentation

Dave Cooper

November 21, 2012

Contents

Chapter 1

Introduction

1.1 Welcome

Congratulations on your purchase or download of Genworks Gendl. By investing some of your valuable time into learning this system, you are investing in your future productivity and you are becoming part of a quiet revolution. Although you may have come to Genworks Gendl because of an interest in 3D modeling or mechanical engineering, you will find that a whole new world, and a whole new approach to computing, will now be at your fingertips.

1.2 Knowledge Base Concepts According to Genworks

You may have an idea about Knowledge Base Systems, or Knowledge *Based* Systems, from college textbooks or corporate marketing propaganda, and found the concept too broad to be of practical use. Or you may have heard jabs at the pretentious-sounding name, "Knowledge-based Engineering," as in: "you mean as opposed to Ignorance-based Engineering?"

To provide a clearer picture, we hope you will agree that our concept of a KB system is simple and practical, and in this tutorial our goal is to make you comfortable and excited about the ideas we have implemented in our flagship system, GenDL (or "Gendl"

Our definition of a *Knowledge Base System* is an object-oriented programming environment which implements the features of *Caching* and *Dependency tracking*. Caching means that once the KB has computed something, it might not need to repeat that computation if the same question is asked again. Dependency tracking is the flip side of that coin — it ensures that if a cached result is *stale*, the result will be recomputed the next time it is *demanded*, so as to give a fresh result.

1.3 Goals for this Tutorial

This manual is designed as a companion to a live two-hour GDL/GWL tutorial, but you may also be reading it on your own. In either case, the basic goals are:

- Get you excited about using GDL/GWL
- Enable you to judge whether GDL/GWL is an appropriate tool for a given job

- Arm you with the ability to argue the case for using GDL/GWL when appropriate
- Prepare you to begin maintaining and authoring GDL/GWL applications, or porting apps from similar KB systems into GDL/GWL.

This manual will begin with an introduction to the Common Lisp programming language. If you are new to Common Lisp: congratulations! You have just discovered a powerful tool backed by a powerful standard specification, which will protect your development investment for decades to come. In addition to the brief overview in this manual, many resources are available to get you started in CL — for starters, we recommend Basic Lisp Techniques[1], which was prepared by the author of this tutorial.

1.4 What is GenDL?

GenDL (or Gendl to be a bit more relaxed) is an acronym for "General-purpose Declarative Language."

GenDL is a superset of ANSI Common Lisp, and consists mainly of automatic code-expanding extensions to Common Lisp implemented in the form of macros. When you write, let's say, 20 lines in GenDL, you might be writing the equivalent of 200 lines of Common Lisp. Of course, since GenDL is a superset of Common Lisp, you still have the full power of the CL language at your fingertips whenever you are working in GenDL.

Since GDL expands into CL, everything you write in GDL will be compiled "down to the metal" to machine code with all the optimizations and safety that the tested-and-true CL compiler provides. This is an important distinction as contrasted to some other so-called KB systems on the market, which are really nothing more than interpreted scripting languages which often impose arbitrary limits on the size and complexity of your application.

GenDL is also a true *declarative* language. When you put together a GDL application, you write and think mainly in terms of objects and their properties, and how they depend on one another in a direct sense. You do not have to track in your mind explicitly how one object or property will "call" another object or propery, in what order this will happen, etc. Those details are taken care of for you automatically by the language.

Because GDL is object-oriented, you have all the features you would normally expect from an object-oriented language, such as

- Separation between the *definition* of an object and an *instance* of an object
- High levels of data abstraction
- The ability for one object to "inherit" from others
- The ability to "use" an object without concern for its "under-the-hood" implementation

GDL supports the "message-passing" paradigm of object orientation, with some extensions. Since full-blown ANSI CLOS (Common Lisp Object System) is always available as well, the Generic

[1] BLT is available at `http://www.franz.com/resources/educational_resources/cooper.book.pdf`

Function paradigm is supported as well. Do not be concerned at this point if you are not fully aware of the differences between these two paradigms[2].

1.5 Why GDL (what is GDL good for?)

- Organizing and interrelating large amounts of information in ways not possible or not practical using conventional languages or conventional relational database technology alone;
- Evaluating many design or engineering alternatives and performing various kinds of optimizations within specified design spaces;
- Capturing the procedures and rules used to solve repetitive tasks in engineering and other fields;
- Applying rules to achieve intermediate and final outputs, which may include virtual models of wireframe, surface, and solid geometric objects.

1.6 What GDL is not

- A CAD system (although it may operate on and/or generate geometric entities);
- A drawing program (although it may operate on and/or generate geometric entities);
- An Artificial Intelligence system (although it is an excellent environment for developing capabilities which could be considered as such);
- An Expert System Shell (although one could be easily embedded within it).

Without further ado, then, let's turn the page and get started with some hands-on GDL...

[2]See Paul Graham's ANSI Common Lisp, page 192, for an excellent discussion of the Two Models of Object-oriented Programming.

Chapter 2

Installation

Follow Section 2.1 if your email address is registered with Genworks and you will install a pre-packaged Gendl distribution including its own Common Lisp engine. Gendl is also available as open-source software[1]; if you want to use that version, then please refer to Section 2.2.

2.1 Installation of pre-packaged Gendl

This section will take you through the installation of Gendl from a prepackaged distribution with the Allegro CL Common Lisp engine and the Slime IDE (based on Gnu Emacs).

2.1.1 Download the Software and retrieve a license key

1. Visit the Downloads section of the Genworks Newsite
2. Enter your email address[2].
3. Download the latest Payload and gpl.zip for Windows[3]
4. Click to receive license key file by email.

2.1.2 Unpack the Distribution

GenDL is currently distributed for all the platforms as a self-contained "zip" file which does not require official administrator installation. What follows are general instructions; more up-to-date details may be found in the email which accompanies the license key file. A five-minute installation video is also available in the Documentation section of the Genworks Newsite.

1. Unzip the gdl1581... zipfile into a location where you have write permissions
2. Unzip the gpl.zip file at the same level as the gdl payload
3. Copy the license key file as gdl.lic (for Trial, Student, Professional editions), or devel.lic (for Enterprise edition) into the `program/` directory within the gdl1581.../ directory.

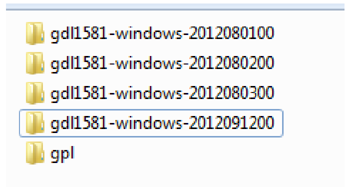

Figure 2.1: Several Gendl versions and one GPL

So you now should have two directories at the same level: one named `gdl1581.../`(the rest of the name will contain the specific dated build stamp), and a `gdl/`directory at the same level. Note that as seen in Figure 2.1, it is possible to have several Gendl versions installed, but just a single common `gpl/` folder.

2.1.3 Make a Desktop Shortcut

1. Using the "My Computer" or "Computer" Windows file manager, right-mouse on the `run-gdl.bat` file.

2. Select "Create Shortcut."

3. Now drag the new "Run-gdl-shortcut" icon to your desktop.

2.1.4 Populate your Initialization File

The default initialization file for Gendl is called `gdlinit.cl`,

2.2 Installation of open-source Gendl

This section is only relevant if you have not received a pre-packaged Gendl distribution with its own Common Lisp engine. If you have received a pre-packaged Gendl distribution, then please skip this section. In case you want to use the open-source Gendl, you will use your own Common Lisp installation and fetch Gendl (Genworks-GDL) using a very powerful and convenient CL package/library manager called *Quicklisp*.

2.2.1 Install and Configure your Common Lisp environment

Gendl is currently tested to build on the following Common Lisp engines:

- Allegro CL (commercial product from Franz Inc, free Express Edition available)
- LispWorks (commercial product from LispWorks Ltd, free Personal Edition available)
- Steel Bank Common Lisp (SBCL) (free open-source project with permissive license)

[1]http://github.com/genworks/Genworks-GDL
[2]if your address is not on file, send mail to licensing@genworks.com
[3]If you already have a gpl.zip from a previous Gendl installation, it is not necessary to download a new one.

Please refer to the documentation for each of these systems for full information on installing and configuring the environment. Typically this will include a text editor, either Gnu Emacs with Superior Lisp Interaction Mode for Emacs (Slime), or a built-in text editing and development environment which comes with the Common Lisp system.

As of this writing, a convenient way to set up Emacs with Slime is to use the Quicklisp-slime-helper.

2.2.2 Load and Configure Quicklisp

As of this writing, Quicklisp is rapidly becoming the defacto standard library manager for Common Lisp.

- Visit the Quicklisp website
- Follow the instructions there to download the `quicklisp.lisp` bootstrap file and load it to set up your Quicklisp environment.

2.3 System Startup and Testing

2.3.1 System Startup

Startup of prepackaged Gendl distribution

To start a prepackaged system, follow these steps:

1. Invoke the `run-gdl.bat` (Windows), or `run-gdl` (Linux, MacOS) startup script. This should launch Gnu Emacs with a README file displayed by default. Take the time to look through this README file. Especially the later part of the file contains information about Emacs keyboard shortcuts available.

2. In emacs, enter: `M-x glime`. That is, hold down the "Meta" (or "Alt") key, and press the "X" key, then type "glime." You will see this command shown in the *mini-buffer* at the bottom of the Emacs window, as shown in Figure 2.2

3. press the "Enter" key

4. On Windows, you will get a new window, named the the Genworks Gendl Console, as shown in Figure 2.3. This window might start out in minimized form (as an icon at the bottom of your screen). Click on it to open it. Watch this console for any errors or warnings.

   The first time you start up, you may see messages in this console for several minutes while the system is being built (or if you received a completely pre-built system, the messages may last only a few seconds).

   On Linux or MacOS, there will be a separate Emacs buffer (available through Emacs' "Buffers" menu) where you will see these messages.

   The messages will consist of compiling and loading information, followed by copyright and welcome information for Gendl. After these messages have finished, you should see the following command prompt:

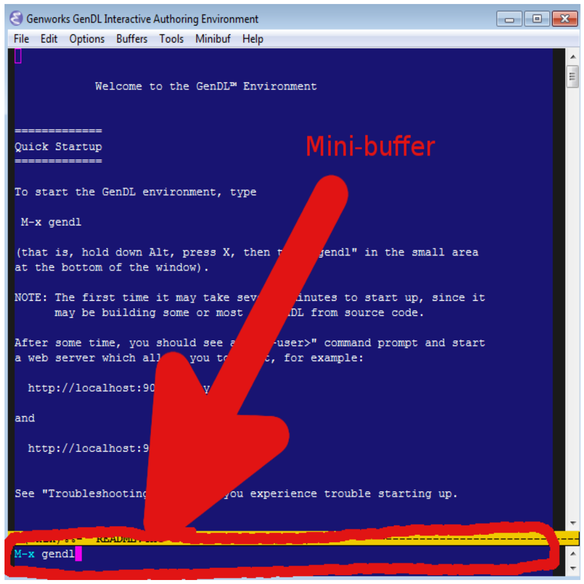

Figure 2.2: The mini-buffer in Emacs

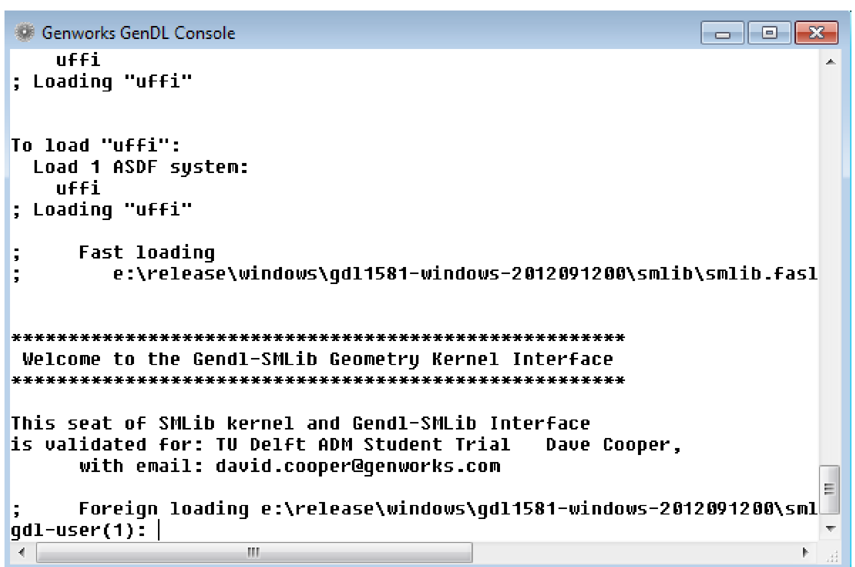

Figure 2.3: Genworks Gendl Console

`gdl-user(1):`

The Genworks GenDL console contains a command prompt, but mostly you will use the *slime-repl...* buffer in Emacs to type commands. The Genworks GenDL console is mainly used for displaying output text from the Gendl system and from your application.

Startup of open-source Gendl distribution

To start an Open-source distribution, follow these steps:

1. Start your Common Lisp engine and development environment (e.g. SBCL with Emacs and Superior Lisp Interaction Mode for Emacs).

2. After Quicklisp is installed and initialized in your system, type: `(ql:quickload :genworks-gdl)` to get Genworks Gendl installed and loaded in your environment.

3. Type the following to initialize the Gendl environment:

   `(gdl:start-gdl :edition :open-source)`

2.3.2 Basic System Test

You may test your installation using the following checklist. These tests are optional. You may perform some or all of them in order to ensure that your Gendl is installed correctly and running smoothly. In your Web Browser (e.g. Google Chrome, Firefox, Safari, Opera, Internet Explorer), perform the following steps:

1. visit http://localhost:9000/tasty.

2. accept default robot:assembly.

3. Select "Add Leaves" from the Tree menu.

4. Click on the top node in the tree.

5. Observe the wireframe graphics for the robot as shown in 2.4.

6. Click on the robot to zoom in.

7. Select "Clear View!" from the View menu.

8. Select "X3DOM" from the View menu.

9. Click on the top node in the tree.

10. "Refresh" or "Reload" your browser window (may not be necessary).

11. If your browser supports WebGL, you will see the robot in shaded dynamic view as shown in Figure 2.5.

12. Select "PNG" from the View menu. You will see the wireframe view of the robot as a PNG image.

13. Select "X3D" from the View menu. If your browser has an X3D plugin installed (e.g. BS Contact), you will see the robot in a shaded dynamic view.

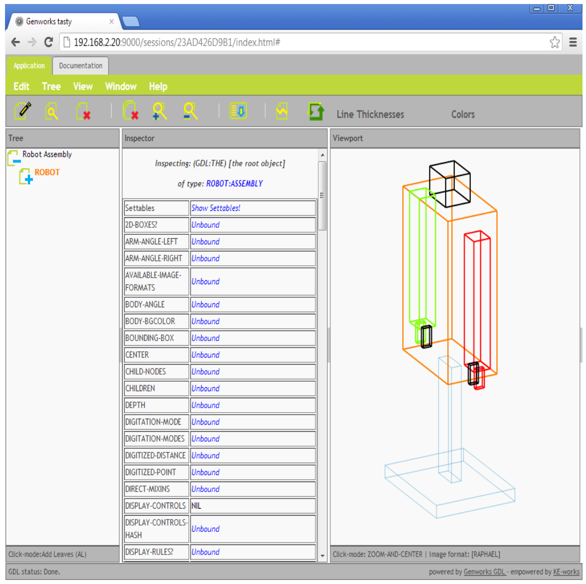

Figure 2.4: Robot displayed in Tasty

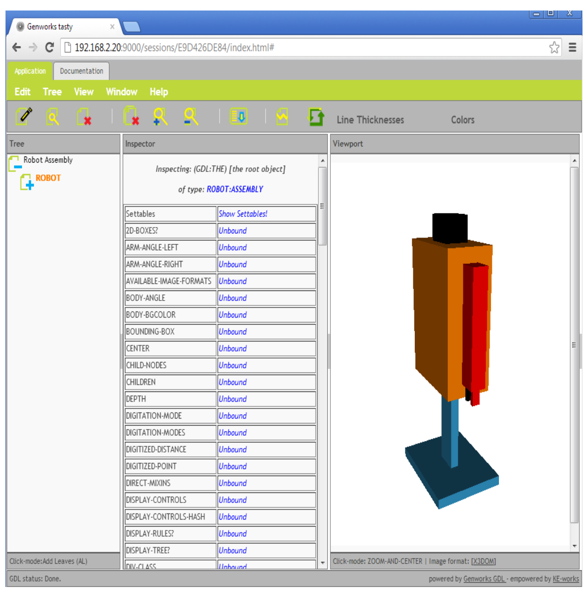

Figure 2.5: Robot x3dom

2.3.3 Full Regression Test

The following commands will invoke a full regression test, including a test of the Surface and Solids primitives provided by the SMLib geometry kernel. Note that the SMLib geometry kernel is only available with proprietary Gendl licenses — therefore if you have an open-source or Trial version, you these regression tests will not all function.

In Emacs at the `gdl-user>` prompt in the `*slime-repl...*` buffer, type the following commands:

1. `(ql:quickload :gdl-regression)`
2. `(gdl-lift-utils::define-regression-tests)`
3. `(gdl-lift-utils::run-regression-tests-pass-fail)`
4. `(pprint gdl-lift-utils::*regression-test-report*)`

2.4 Getting Help and Support

If you run into unexplained errors in the installation and startup process, please contact the following resources:

1. Make a posting to the Genworks Google Group
2. For pure Common Lisp issues, join the #lisp IRC (Internet Relay Chat) channel and discuss issues there.
3. Also for Common Lisp issues, follow the comp.lang.lisp Usenet group.
4. If you are a supported Genworks customer, send email to support@genworks.com
5. If you are not a supported Genworks customer but you want to report an apparent bug or have other suggestions or inquiries, you may also send email to support@genworks.com, but please understand that Genworks cannot guarantee any response or a particular timeframe for any response.

Chapter 3

Basic Operation of the Gendl Environment

This chapter will step you through all the basic steps of operating a typical Gendl environment. We will not go into any depth about the additional features of the environment or language syntax in this section — this is just for getting familiar and practicing with the mechanics of operating the environment with a keyboard.

3.1 What is Different about Gendl?

Gendl is a dynamic language environment with incremental compiling and in-memory definitions. That means that as long as the system is running, you can *compile* new *definitions* of functions, objects, etc, and they will immediately become available as part of the running system, and you can begin testing them immediately or update an existing set of objects to observe their new behavior.

In many other programming language systems, you have to start the system from the beginning and reload all the files in order to test new functionality.

In Gendl, if you simply shut down the system after having compiled and loaded a set of files with new definitions, then when you restart the system you will have to recompile and/or reload those definitions in order to bring the system back into the same state. This is typically done automatically, using commands placed into the `gdlinit.cl` initialization file, as introduced in Section 3.4. Alternatively, you can compile and load definitions into your Gendl session, then save the "world" in that state. That way, it is possible to start a new Gendl "world" which already has all your application's definitions loaded and ready for use, without having to procedurally reload any files. You can then begin to make and test new definitions (and re-definitions) starting from this new "world."

3.2 Startup, "Hello, World!" and Shutdown

The typical Gendl environment consists of three programs: Gnu Emacs (the editor), a Common Lisp engine with Gendl system loaded or built into it (e.g. the `gdl.exe` executable in your `program/` directory), and (optionally) a web browser such as Firefox, Google Chrome, Safari, Opera, or Internet Explorer. Emacs runs as the main *process*, and this in turn starts the CL engine with

Gendl as a *sub-process*. The CL engine typically runs an embedded *webserver*, enabling you to access your application through a standard web browser.

As introduced in Chapter 2, the typical way to start a pre-packaged Gendl environment is with the `run-gdl.bat` (Windows), or `run-gdl` (MacOS, Linux) script files. Invoke this script file from your computer's file manager, or from a desktop shortcut if you have created one as outlined in section 2.1.3. Your installation executable may also have created a Windows "Start" menu item for Genworks Gendl. Of course you can also invoke `run-gdl.bat` from the Windows "cmd" command-line, or from another command shell such as Cygwin.[1]

3.2.1 Startup

Startup of a typical Gendl development session consists of two fundamental steps: (1) starting the Emacs editing environment, and (2) starting the actual Gendl process as a "sub-process" or "inferior" process within Emacs. The Gendl process should automatically establish a network connection back to Emacs, allowing you to interact directly with the Gendl process from within Emacs.

1. Invoke the `run-gdl.bat` or `run-gdl.bat` startup script.

2. You should see a blue emacs window as in Figure **??**. (alternative colors are also possible).

3. Press M-x (Alt-x), and type `gendl` in the mini-buffer, as seen in Figure 2.2.

4. (MS Windows): Look for the Genworks Gendl Console window, or (Linux, Mac) use the Emacs "Buffer" menu to visit the "*inferior-lisp*" buffer. Note that the Genworks Gendl Console window might start as a minimized icon; click or double-click it to un-minimize.

5. Watch the Genworks GDL Console window for any errors. Depending on your specific installation, it may take from a few seconds to several minutes for the Genworks Gendl Console (or *inferior-lisp* buffer) to settle down and give you a `gdl-user():` prompt. This window is where you will see most of your program's textual output, any error messages, warnings, etc.

6. In Emacs, type: `C-x &` (or select Emacs menu item Buffers→*slime-repl...*) to visit the "*slime-repl ...*" buffer. The full name of this buffer depends on the specific CL/Gendl platform which you are running. This buffer contains an interactive prompt, labeled `gdl-user>`, where you will enter most of your commands to interact with your running Gendl session for testing, debugging, etc. There is also a web-based graphical interactive environment called *tasty* which will will cover in Chapter **??**

7. To ensure that the Gendl interpreter is up and running, type: `(+ 2 3)` and press [Enter].

8. You should see the result `5` echoed back to you below the prompt.

[1]Cygwin is also useful as a command-line tool on Windows for interacting with a version control system like Subversion (svn).

```
(in-package :gdl-user)

(define-object hello ()

  :computed-slots
  ((greeting "Hello, World!")))
```

Figure 3.1: Example of Simple Object Definition

3.2.2 Developing and Testing a Gendl "Hello World" application

1. type C-x (Control-x) 2, or C-x 3, or use the "Split Screen" option of the File menu to split the Emacs frame into two "windows" ("windows" in Emacs are non-overlapping panels, or rectangular areas within the main Emacs window).

2. type C-x o several times to move from one window to the other, or move the mouse cursor and click in each window. Notice how the blinking insertion point moves from one window to the other.

3. In the top (or left) window, type C-x C-f (or select Emacs menu item "File→Open File") to get the "Find file" prompt in the mini-buffer.

4. Type C-a to move the point to the beginning of the mini-buffer line.

5. Type C-k to delete from the point to the end of the mini-buffer.

6. Type `~/hello.gdl` and press [Enter]

7. You are now editing a (presumably new) file of Gendl code, located in your HOME directory, called `hello.gdl`

8. Enter the text from Figure 3.1 into the `hello.gdl` buffer. You do not have to match the line breaks and whitespace as shown in the example. You can auto-indent each new line by pressing [TAB] after pressing [Enter] for the newline.

   *Protip:*You can also try using `C-j` instead of [Enter], which will automatically give a newline and auto-indent.

9. type `C-x C-s` (or choose Emacs menu item *File→Save*) to save the contents of the buffer (i.e. the window) to the file in your HOME directory.

10. type `C-c C-k` (or choose Emacs menu item *SLIME→Compilation→Compile/Load File*) to compile & load the code from this file.

11. type `C-c o` (or move and click the mouse) to switch to the bottom window.

12. In the bottom window, type `C-x &` (or choose Emacs menu item *Buffers→*slime-repl...**) to get the `*slime-repl ...*` buffer, which should contain a `gdl-user>` prompt. This is where you normally type interactive Gendl commands.

13. If necessary, type `M >` (that is, hold down Meta (Alt), Shift, and the ">" key) to move the insertion point to the end of this buffer.

14. At the `gdl-user>` prompt, type

    ```
    (make-self 'hello)
    ```

    and press [Enter].

15. At the `gdl-user>` prompt, type

    ```
    (the greeting)
    ```

    and press [Enter].

16. You should see the words `Hello, World!` echoed back to you below the prompt.

3.2.3 Shutdown

To shut down a development session gracefully, you should first shut down the Gendl process, then shut down your Emacs.

- Type `M-x quit-gendl` (that is, hold Alt and press X, then release both while you type `quit-gendl` in the mini-buffer), then press [Enter]

- Type `C-x C-c` to quit from Emacs. Emacs will prompt you to save any modified buffers before exiting.

3.3 Working with Projects

Gendl contains utilities which allow you to treat your application as a "project," with the ability to compile, incrementally compile, and load a "project" from a directory tree of source files representing your project. In this section we give an overview of the expected directory structure and available control files, followed by a reference for each of the functions included in the bootstrap module.

3.3.1 Directory Structure

You should structure your applications in a modular fashion, with the directories containing actual Lisp sources called "source."

You may have subdirectories which themselves contain "source" directories.

We recommend keeping your codebase directories relatively flat, however.

In Figure 3.2 is an example application directory, with four source files.

```
apps/yoyodyne/booster-rocket/source/assembly.gdl
apps/yoyodyne/booster-rocket/source/package.gdl
apps/yoyodyne/booster-rocket/source/parameters.gdl
apps/yoyodyne/booster-rocket/source/rules.gdl
```

Figure 3.2: Example project directory with four source files

```
apps/yoyodyne/booster-rocket/source/assembly.gdl
apps/yoyodyne/booster-rocket/source/file-ordering.isc
apps/yoyodyne/booster-rocket/source/package.gdl
apps/yoyodyne/booster-rocket/source/parameters.gdl
apps/yoyodyne/booster-rocket/source/rules.gdl
```

Figure 3.3: Example project directory with file ordering configuration file

3.3.2 Source Files within a source/ subdirectory

Enforcing Ordering

Within a source subdirectory, you may have a file called `file-ordering.isc`[2] to enforce a certain ordering on the files. Here is the contents of an example for the above application:

```
("package" "parameters")
```

This will force package.lisp to be compiled/loaded first, and parameters.lisp to be compiled/loaded next. The ordering on the rest of the files should not matter (although it will default to lexigraphical ordering).

Now our sample application directory looks like Figure 3.3 is an example application directory, with four source files.

3.3.3 Generating an ASDF System

ASDF stands for Another System Definition Facility, which is the predominant system in use for Common Lisp third-party libraries. With Gendl, you can use the `:create-asd-file?` keyword argument to make cl-lite generate an ASDF system file instead of actually compiling and loading the system. For example:

```
(cl-lite "apps/yoyodyne/" :create-asd-file? t)
```

In order to include a depends-on clause in your ASDF system file, create a file called `depends-on.isc` in the toplevel directory of your system. In this file, place a list of the systems your system depends on. This can be systems from your own local projects, or from third-party libraries. For example, if your system depends on the `:cl-json` third-party library, you would have the following contents in your `depends-on.isc`:

```
(:cl-json)
```

[2] `isc` stands for "Intelligent Source Configuration"

3.3.4 Compiling and Loading a System

Once you have generated an ASDF file, you can compile and load the system using Quicklisp. To do this for our example, follow these steps:

1. `(cl-lite "apps/yoyodyne/" :create-asd-file? t)`

   to generate the asdf file for the yoyodyne system. This only has to be done once after every time you add, remove, or rename a file or folder from the system.

2. `(pushnew "apps/yoyodyne/" ql:*local-project-directories*)`

   This can be done in your `gdlinit.cl` for projects you want available during every development session. Note that you should include the full path prefix for the directory containing the ASDF system file.

3. `(ql:quickload :gdl-yoyodyne)`

   this will compile and load the actual system. Quicklisp uses ASDF at the low level to compile and load the systems, and Quicklisp will fetch any depended-upon third-party libraries from the Internet on-demand. Source files will be compiled only if the corresponding binary (fasl) file does not exist or is older than the source file. By default, ASDF keeps its binary files in a*cache* directory, separated according to CL platform and operating system. The location of this cache is system-dependent, but you can see where it is by observing the compile and load process.

3.4 Customizing your Environment

You may customize your environment in several different ways, for example by loading definitions and settings into your Gendl "world" automatically when the system starts, and by specifying fonts, colors, and default buffers (to name a few) for your emacs editing environment.

3.5 Saving the World

Saving the world refers to a technique of saving a complete binary image of your Gendl "world" which contains all the currently compiled and loaded definitions and settings. This allows you to start up a saved world almost instantly, without having to reload all the definitions. You can then incrementally compile and load just the particular definitions which you are working on for your development session.

To save a world, follow these steps:

1. Load the base Gendl code and (optionally) code for Gendl modules (e.g. gdl-yadd, gdl-tasty) you want to be in your saved image. For example:

   ```
   (ql:quickload :gdl-yadd)
   (ql:quickload :gdl-tasty)
   ```

2. `(ff:unload-foreign-library (merge-pathnames "smlib.dll" "sys:smlib;"))`

3. `(net.aserve:shutdown)`
4. `(setq excl:*restart-init-function* '(gdl:start-gdl :edition :trial))`
5. (to save an image named yoyodyne.dxl) Invoke the command

   ```
   (dumplisp :name "yoyodyne.dxl")
   ```

   Note that the standard extension for Allegro CL images is `.dxl`. Prepend the file name with path information, to write the image to a specific location.

3.6 Starting up a Saved World

In order to start up Gendl using a custom saved image, or "world," follow these steps

1. Exit Gendl
2. Copy the supplied `gdl.dxl` to `gdl-orig.dxl`.
3. Move the custom saved dxl image to `gdl.dxl` in the Gendl application "program/" directory.
4. Start Gendl as usual. Note: you may have to edit the system gdlinit.cl or your home gdlinit.cl to stop it from loading redundant code which is already in the saved image.

Chapter 4

Understanding Common Lisp

Gendl is a superset of Common Lisp, and is embedded in Common Lisp. This means that when working with Gendl, you have the full power of CL available to you. The lowest-level expressions in a Gendl definition are CL "symbolic expressions," or "s-expressions." This chapter will familiarize you with CL s-expressions.

4.1 S-expression Fundamentals

S-expressions can be used in your definitions to establish the value of a particular *slot* in an object, which will be computed on-demand. You can also evaluate S-expressions at the toplevel `gdl-user`> prompt, and see the result immediately. In fact, this toplevel prompt is called a *read-eval-print* loop, because its purpose is to *read* each s-expression entered, *evaluate* the expression to yield a result (or *return-value*), and finally to *print* that result.

CL s-expressions use a *prefix* notation, which means that they consist of either an *atom* (e.g. number, text string, symbol) or a list (one or more items enclosed by parentheses, where the first item is taken as a symbol which names an operator). Here is an example:

```
(+ 2 2)
```

This expression consists of the function named by the symbol `+`, followed by the numeric arguments `2` and another `2`. As you may have guessed, when this expression is evaluated it will return the value 4. *Try it:* try typing this expression at your command prompt, and see the return-value being printed on the console. What is actually happening here? When CL is asked to evaluate an expression, it processes the expression according to the following rules:

- If the expression is an *atom* (e.g. a non-list datatype such as a number, text string, or literal symbol), it simply evaluates to itself. Examples:

  Note that numbers are represented directly (with decimal points and slashes for fractions allowed), strings are surrounded by double-quotes, and literal symbols are introduced with a preceding single-quote. Symbols are allowed to have dashes ("-") and most other special characters. By convention, the dash is used as a word separator in CL symbols.

- If the expression is a list (i.e. is surrounded by parentheses), CL processes the *first* element in this list as an *operator name*, and the *rest* of the elements in the list represent the *arguments* to the operator. An operator can take zero or more arguments, and can return zero or more return-values. Some operators evaluate their arguments immediately and work directly on those values (these are called *functions*. Other operators expand into other code. These are called *special operators* or *macros*. Macros are what give Lisp (and CL in particular) its special power. Here are some examples of functional s-expressions:
  - `gdl-user> (expt 2 5)`
    `32`
  - `gdl-user> (+ 2 5)`
    `7`
  - `gdl-user> (+ 2)`
    `2`
  - `gdl-user> (+ (+ 2 2) (+ 3 3 ))`
    `10`

4.2 Fundamental CL Data Types

As we have seen, Common Lisp natively supports many data types common to other languages, such as numbers and text strings. CL also contains several compound data types such as lists, arrays, and hash tables. CL contains *symbols* as well, which typically are used as names for other data elements.

Regarding data types, CL follows a paradigm called dynamic typing. Basically this means that values have type, but variables do not necessarily have type, and typically variables are not "pre-declared" to be of a particular type.

4.2.1 Numbers

As we have seen, numbers in CL are a native data type which simply evaluate to themselves when entered at the toplevel or included in an expression.

Numbers in CL form a hierarchy of types, which includes Integers, Ratios, Floating Point, and Complex numbers. For many purposes, you only need to think of a value as a "number" without getting any more specific than that. Most arithmetic operations, such as `+`, `-`, `*`, `/` etc, will automaticaly do any necessary type coercion on their arguments and will return a number of the appropriate type.

CL supports a full range of floating-point decimal numbers, as well as true Ratios, which means that `1/3` is a true one-third, not `0.333333333` rounded off at some arbitrary precision.

4.2.2 Strings

Strings are actually a specialized kind of array, namely a one-dimensional array (vector) made up of text characters. These characters can be letters, numbers, or punctuation, and in some cases can include characters from international character sets (e.g. Unicode or UTF-8) such as Chinese Hanzi or Japanese Kanji. The string delimiter in CL is the double-quote character.

As we have seen, strings in CL are a native data type which simply evaluate to themselves when included in an expression.

4.2.3 Symbols

Symbols are such an important data structure in CL, that people sometimes refer to CL as a "Symbolic Computing Language." Symbols are a type of CL object which provides your program with a built-in mechanism to store and retrieve values and functions, as well as being useful in their own right. A symbol is most often known by its name (actually a string), but in fact there is much more to a symbol than its name. In addition to the name, symbols also contain a *function* slot, a *value* slot, and an open-ended *property-list* slot in which you can store an arbitrary number of named properties.

For a named function such as `+` the function-slot contains the actual function object for performing numeric addition. The value-slot of a symbol can contain any value, allowing the symbol to act as a global variable, or *parameter*. And the property-list, also known as the *plist* slot, can contain an arbitrary amount of information.

This separation of the symbol data structure into function, value, and plist slots is one obvious distinction between Common Lisp and most other Lisp dialects. Most other dialects allow only one (1) "thing" to be stored in the symbol data structure, other than its name (e.g. either a function or a value, but not both at the same time). Because Common Lisp does not impose this restriction, it is not necessary to contrive names, for example for your variables, to avoid conflicting with existing "reserved words" in the system. For example, `list` is the name of a built-in function in CL. But you may freely use `list` as a variable name as well. There is no need to contrive arbitrary abbreviations such as `lst`.

How symbols are evaluated depends on where they occur in an expression. As we have seen, if a symbol appears first in a list expression, as with the `+` in `(+ 2 2)`, the symbol is evaluated for its function slot. If the first element of an expression indeed has a function in its function slot, then any subsequent symbol in the expression is taken as a variable, and it is evaluated for its global or local value, depending on its scope (more on variables and scope later).

As noted in Section 3.1.3, if you want a literal symbol itself, one way to achieve this is to "quote" the symbol name:

```
'a
```

Another way is for the symbol to appear within a quoted list expression, for example:

```
'(a b c)
```

or

```
'(a (b c) d)
```

Note that the quote (`'`) applies across everything in the list expression, including any sub-expressions.

4.2.4 Lists

Lisp takes its name from its strong support for the list data structure. The list concept is important to CL for more than this reason alone — most notably, lists are important because *all CL programs are themselves lists.*

Having the list as a native data structure, as well as the form of all programs, means that it is straightforward for CL programs to compute and generate other CL programs. Likewise, CL programs can read and manipulate other CL programs in a natural manner. This cannot be said of most other languages, and is one of the primary distinguishing characteristics of Lisp as a language.

Textually, a list is defined as zero or more elements surrounded by parentheses. The elements can be objects of any valid CL data types, such as numbers, strings, symbols, lists, or other kinds of objects. As we have seen, you must quote a literal list to evaluate it or CL will assume you are calling a function. Now look at the following list:

```
(defun hello () (write-string "Hello, World!"))
```

This list also happens to be a valid CL program (function definition, in this case). Don't worry about analyzing the function definition right now, but do take a few moments to convince yourself that it meets the requirements for a list.

What are the types of the elements in this list?

In addition to using the quote (') to produce a literal list, another way to produce a list is to call the function `list`. The function `list` takes any number of arguments, and returns a list made up from the result of evaluating each argument. As with all functions, the arguments to the `list` function get evaluated, from left to right, before being processed by the function. For example:

```
(list a b (+ 2 2))
```

will return the list

```
(a b 4)
```

. The two quoted symbols evaluate to symbols, and the function call `(+ 2 2)` evaluates to the number 4.

Chapter 5

Understanding Gendl — Core GDL Syntax

Now that you have a basic grasp of Common Lisp syntax (or, more accurately, *lack* of syntax), we will jump directly into the Gendl framework. By using Gendl you can formulate most of your engineering and computing problems in a natural way, without becoming bogged down in the complexity of the Common Lisp Object System (CLOS).

The Gendl product is a commercially available KBE system dual-licensed under the Affero Gnu Public License and a Proprietary license. The core GDL language is a proposed standard for a vendor-neutral KBE language.

As we mentioned in the previous chapter, Gendl is based on and is a superset of ANSI Common Lisp. Because ANSI CL is an unencumbered, open standard with several commercial and free implementations, it is a good bet that applications written in it will continue to function 50, 100, or even hundreds of years from now.

Note that the historical name of Gendl was "GDL," and this name persists throughout the product for example for naming Common Lisp packages.

5.1 Defining a Working Package

In Common Lisp, *packages* are a mechanism to separate symbols into namespaces. Using packages it is possible to avoid naming collisions in large projects. Consider this analogy: in the United States, telephone numbers consist of a three-digit area code and a seven-digit number. The same seven-digit number can occur in two or more separate area codes, without causing a conflict.

The macro `gdl:define-package`is used to set up a new working package in Gendl.

Example:

```
(gdl:define-package :yoyodyne)
```

will establish a new package called `:yoyodyne` which has all the Gendl operators available.

The `:gdl-user` package is an empty, pre-defined package for your use if you do not wish to make a new package just for scratch work.

For real projects it is recommended that you make and work in your own Gendl package, defined as above with `gdl:define-package`.

*Notes for advanced users:*Packages defined with `gdl:define-package` will implicitly *use* the `:gdl` package and the `:common-lisp` package, so you will have access to all exported symbols in these packages without prefixing them with their package name.

You may extend this behavior, by calling `gdl:define-package` and adding additional packages to use with `(:use ...)`. For example, if you want to work in a package with access to GDL operators, Common Lisp operators, and symbols from the `:cl-json` package [1], you could set it up as follows:

```
(ql:quickload :cl-json)
(gdl:define-package :yoyodyne (:use :cl-json))
```

. the first form ensures that the cl-json code module is actually fetched and loaded. The second form defines a package with the `:cl-json` operators available to it.

5.2 Define-Object

Define-object is the basic macro for defining objects in GDL. An object definition maps directly into a Lisp (CLOS) class definition.

The `define-object` macro takes three basic arguments:

- a *name*, which is a symbol;
- a *mixin-list*, which is a list of symbols naming other objects from which the current object will inherit characteristics;
- a *specification-plist*, which is spliced in (i.e. doesn't have its own surrounding parentheses) after the mixin-list, and describes the object model by specifying properties of the object (messages, contained objects, etc.) The specification-plist typically makes up the bulk of the object definition.

Here are descriptions of the most common keywords making up the specification-plist:

input-slots specify information to be passed into the object instance when it is created.

computed-slots are really cached methods, with expressions to compute and return a value.

objects specify other instances to be "contained" within this instance.

functions are (uncached) functions "of" the object, i.e. they are actually methods which discriminate on their first argument, which is the object instance upon which they are operating. GDL functions can also take other non-specialized arguments, just like a normal CL function.

Figure 5.1 shows a simple example, which contains two input-slots, `first-name` and `last-name`, and a single computed-slot, `greeting`. As you can see, a GDL Object is analogous in some ways to a `defun`, where the input-slots are like arguments to the function, and the computed-slots are

[1] CL-JSON is a free third-party library for handling JSON format, a common data format used for Internet applications.

```
(define-object hello ()
  :input-slots (first-name last-name)

  :computed-slots
  ((greeting (format nil "Hello, ~a ~a!!"
                     (the first-name)
                     (the last-name)))))
```

Figure 5.1: Example of Simple Object Definition

like return-values. But seen another way, each attribute in a GDL object is like a function in its own right.

The referencing macro `the` shadows CL's `the` (which is a seldom-used type declaration operator). `The` in GDL is a macro which is used to reference the value of other messages within the same object or within contained objects. In the above example, we are using `the` to refer to the values of the messages (input-slots) named `first-name` and `last-name`.

Note that messages used with `the` are given as symbols. These symbols are unaffected by the current Lisp `*package*`, so they can be specified either as plain unquoted symbols or as keyword symbols (i.e. preceded by a colon), and the `the` macro will process them appropriately.

5.3 Making Instances and Sending Messages

Once we have defined an object such as the example above, we can use the constructor function `make-object` in order to create an *instance* of it. This function is very similar to the CLOS `make-instance` function. Here we create an instance of `hello` with specified values for `first-name` and `last-name` (the required input-slots), and assign this instance as the value of the symbol `my-instance`:

```
GDL-USER(16): (setq my-instance
                (make-object 'hello :first-name "John"
                                    :last-name "Doe"))
#<HELLO @ #x218f39c2>
```

As you can see, keyword symbols are used to "tag" the input values, and the return value is an instance of class `hello`. Now that we have an instance, we can use the macro `the-object` to send messages to this instance:

```
GDL-USER(17): (the-object my-instance greeting)
"Hello, John Doe!!"
```

`The-object` is similar to `the`, but as its first argument it takes an expression which evaluates to an object instance. `The`, by contrast, assumes that the object instance is the lexical variable `self`, which is automatically set within the lexical context of a `define-object`.

```
(define-object city ()
  :computed-slots
  ((total-water-usage (+ (the hotel water-usage)
                         (the bank water-usage))))
  :objects
  ((hotel :type 'hotel
          :size :large)
   (bank  :type 'bank
          :size :medium)))
```

Figure 5.2: Object Containing Child Objects

Like `the`, `the-object` evaluates all but the first of its arguments as package-immune symbols, so although keyword symbols may be used, this is not a requirement, and plain, unquoted symbols will work just fine.

For convenience, you can also set `self` manually at the CL Command Prompt, and use `the` instead of `the-object` for referencing:

```
GDL-USER(18): (setq self
                (make-object 'hello :first-name "John"
                                    :last-name "Doe"))
#<HELLO @ #x218f406a>

GDL-USER(19): (the greeting)
"Hello, John Doe!!"
```

In actual fact, `(the ...)` simply expands into `(the-object self ...)`.

5.4 Objects

The `:objects` keyword specifies a list of "contained" instances, where each instance is considered to be a "child" object of the current object. Each child object is of a specified type, which itself must be defined with `define-object` before the child object can be instantiated.

Inputs to each instance are specified as a plist of keywords and value expressions, spliced in after the object's name and type specification. These inputs must match the inputs protocol (i.e. the input-slots) of the object being instantiated. Figure 5.2 shows an example of an object which contains some child objects. In this example, `hotel` and `bank` are presumed to be already (or soon to be) defined as objects themselves, which each answer the `water-usage` message. The *reference chains*:

```
(the hotel water-usage)
```

and

```
(defparameter *presidents-data*
    '((:name
       "Carter"
       :term 1976)
      (:name "Reagan"
       :term 1980)
      (:name "Bush"
       :term 1988)
      (:name "Clinton"
       :term 1992)))

(define-object presidents-container ()

  :input-slots
  ((data *presidents-data*))

  :objects
  ((presidents :type 'president
               :sequence (:size (length (the data)))
               :name (getf (nth (the-child index) (the data)) :name)
               :term (getf (nth (the-child index) (the data)) :term))))
```

Figure 5.3: Sample Data and Object Definition to Contain U.S. Presidents

```
(the bank water-usage)
```

provide the mechanism to access messages within the child object instances.

These child objects become instantiated *on demand*, meaning that the first time these instances or any of their messages are referenced, the actual instance will be created *and* cached for future reference.

5.5 Sequences of Objects and Input-slots with a Default Expression

Objects may be *sequenced*, to specify, in effect, an array or list of object instances. The most common type of sequence is called a *fixed size* sequence. See Figure 5.3 for an example of an object which contains a sequenced set of instances representing U.S. presidents. Each member of the sequenced set is fed inputs from a list of plists, which simulates a relational database table (essentially a "list of rows").

Note the following from this example:

- In order to sequence an object, the input keyword `:sequence` is added, with a list consisting of the keyword `:size` followed by an expression which must evaluate to a number.

- In the input-slots, `data` is specified together with a default expression. Used this way, input-slots function as a hybrid of computed-slots and input-slots, allowing a *default expression* as with computed-slots, but allowing a value to be passed in on instantiation or from the parent, as with an input-slot which has no default expression. A passed-in value will override the default expression.

5.6 Summary

This GDL syntax overview has been kept purposely brief, covering the fundamentals of the language in a dense manner. On one hand, it is not meant to be a comprehensive language reference; on the other hand, do not be concerned if you are still unsure about some of the terminology. The upcoming examples will revisit and further expand many of the topics covered here, and at some point a coherent picture should begin to emerge.

At that point it will be like riding a bicycle, and there will be no going back.

Chapter 6

More Common Lisp for Gendl

Chapter 7

Advanced Gendl

Upgrade Notes

GDL 1580 marked the end of a major branch of GDL development, and 1581 was actually a major new version. With 1581, an open-source version was released, and eventually the name was changed to Gendl.

This chapter lists the typical modifications you will want to consider for upgrading from GDL 1580 to Gendl 1582.

- (update-gdl ..) is not yet available for 1582. Instead of updating incrementally with patches, Gendl 1582 is released on a monthly basis in conjunction with Quicklisp releases. Updating quicklisp involves downloading a full Genworks source code tree and running a build script. Information on this procedure is provided in Section **??**.

- (make-gdl-app ..) not yet available for 1582. We are preparing the Enterprise Edition of 1582 which will include the make-gdl-app function, which creates Runtime applications without the compiler or Gendl development facilities. If you are an Enterprise licensee, are ready to release Runtime applications on 1582, and you have not received information on the Enterprise Edition, please contact support@genworks.com

- (register-asdf-systems) and the "3rdpty/" directory are no longer needed or available. Instead, we depend on the Quicklisp system. Details of Quicklisp are available at . See Section **??** for information about how to use Quicklisp with Gendl.

- There is a system-wide gdlinit.cl in the application directory, and this may have some default information which ships with Gendl. There is a personal one in home directory, which you should modify if you want to customize anything.

- Slime debugging is different from the ELI emacs debugger. The main thing to know is to press "a" or "q" to pop out of the current error. Full documentation for the Slime debug mode is available with the Slime documentation.

- color-themes – Gendl now ships with the Emacs color-theme package. You can select a different color theme with `M-x color-theme-select`. Press [Enter] or middle-mouse on a color theme to apply it.

- Gendl files can now end with `.lisp` or `.gdl`. The new `.gdl` extension will work for emacs Lisp mode and will work with cl-lite, ASDF, and Quicklisp for including source files in application systems. We recommend migrating to the new `.gdl` extension for files containing `define-object`, `define-format`, and `define-lens` forms, and any other future toplevel defining forms introduced by Gendl, in order to distinguish from files containing raw Common Lisp code.

- in gdlAjax, HTML for a sheet-section is given in the slot called `inner-html` instead of `main-view`. This name change was made to clarify what exactly is expected in this slot – it is the innerHTML of the page division represented by the current sheet-section. If you want to make your code back-compatible with GDL 1580, you can use the following form in place of old occurences of `main-view`:

```
... #+allegro-v8.1 main-view #-allegro-v8.1 inner-html ...
```

Index