`ASDF-Literate` v0.1 - Pamphlets in ASDF.

Clifford Yapp and Stephen Wilson

December 21, 2007

Abstract

ASDF, the most widely used system definition utility for Lisp, does not have any default knowledge of how to handle pamphlet files. This file defines the basic routines needed for pamphlet handling and implements methods to load Lisp pamphlets. Separate methods are needed for other pamphlet types.

Contents

1 Copyright and License **3**

2 Design **3**
- 2.1 Pamphlet Centric Definitions 4
- 2.2 Chunk Centric Definitions 4
- 2.3 Hybrid System Definitions 5

3 File Naming and Placement **5**
- 3.1 Setting up Global Variables 6
- 3.2 Priority of Settings 7
- 3.3 The Literate-Settings Class 7
- 3.4 The Literate-System Class 8

4 Basic Setup - Pamphlet Classes **8**
- 4.1 Pathname Commands for Pamphlet Classes 8
- 4.2 General Literate Classes 9
- 4.3 Pathname Commands for asdf-chunk Classes 9
- 4.4 Language Specific Literate Classes 10
- 4.5 Automatic Generation of Output File Names 11

5 Chunk Name Extraction from Component Definitions **12**
- 5.1 Chunks as Top Level Components 12
- 5.2 Chunks in Modules 13

6 Operations **13**

7 Perform - Pamphlets as Components **14**
- 7.1 Compile 15
- 7.2 Load 17
- 7.3 Load-Source 18
- 7.4 Document 18
- 7.5 Build 19

8 Completed Operations **20**

1 Copyright and License

This program is available under the Modified BSD license. The wording of the file was changed slightly, as there is no organization associated with the creation of this file, but if there is any question the license is intended to be Modified BSD.

⟨ Copyright 1 ⟩ ≡

```
;; Copyright (c) 2007, Clifford Yapp, Stephen Wilson
;; All rights reserved.
```

Included by 55.

⟨ License 2 ⟩ ≡

```
;; Redistribution and use in source and binary forms, with or without
;; modification, are permitted provided that the following conditions are
;; met:
;;
;;    - Redistributions of source code must retain the above copyright
;;      notice, this list of conditions and the following disclaimer.
;;
;;    - Redistributions in binary form must reproduce the above copyright
;;      notice, this list of conditions and the following disclaimer in
;;      the documentation and/or other materials provided with the
;;      distribution.
;;
;;    - The names of the contributors may not be used to endorse or promote
;;      products derived from this software without specific prior written
;;      permission.
;;
;;THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
;;IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
;;TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
;;PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
;;OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
;;EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
;;PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
;;PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
;;LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
;;NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
;;SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Included by 55.

2 Design

Another System Definition Facility (a.k.a ASDF) is the most widely used Lisp utility for defining the relationships between components of large systems. Among its benefits is a very significant degree of flexibility - it makes use of the Common Lisp Object System (CLOS) to abstract many of the details of handling software systems.

Axiom, as a literate program, cannot directly make use of ASDF in its default form. Pamphlet files are not loadable in their un-tangled form, and must first be tangled using a WEB type tool. Fortunately, ASDF provides mechanisms to customize its functionality, allowing literate program developers to transparently work with pamphlet files in a Lisp environment.

How pamphlet files are structured makes a difference when defining what constitutes “best practice” description scemantics. It is possible to have a large collection of individual pamphlet files with only a small number of concepts delt with per pamphlet" - in ASDF terms, one (or a small number of) individual system components per file. The other extreme is also possible - a single pamphlet file with a large number of concepts, which would logically constitute many individual ASDF components. Between these endpoints are intermediate documents where either style might be appropriate.

2.1 Pamphlet Centric Definitions

This case is more like the typical non-literate ASDF approach, and so the behaviors that address it are similar to ASDF itself. In general, for the purposes of defining literate source code loads in ASDF the most important design constraint to keep in mind is ASDF's requirement for a unique name for each component within a system. Normally this defaults to the name of the file being loaded, but in the case of pamphlet files the same source file may be used multiple times as different chunks are loaded.

To accomidate this, ASDF-Literate will work with small-file-based systems using a hierarchical order of nominclature:

- (:pamphlet "file") - in the case where the intent is to load the "*" chunk from a file, only the pamphlet file need be specified.
- (:pamphlet "chunkname" :pathname "file") - if a non-default chunk name is needed, the component name becomes the chunk name and the file is specified via the pathname argument.
- (:pamphlet "file-chunkname" :pathname "file" :chunk "chunkname") - in the most general case, when multiple files that are part of the same defsystem use the same chunk name, a unique component name must be provided and both file and chunk name must be explicitly specified. The convention will be to use "file:chunkname" as the component name on the assumption that this is guaranteed to be unique within a defsystem, but it is only a convention - the specified pathname and chunk name will be used by the methods. The name is this case serves only as a unique identifier.[1]

A system definition in this scheme would take the form (for this example Lisp pamphlets are assumed):

```
(defsystem SYSTEM
  :class literate-system
  :components
  ((:cl-pamphlet "FILE1.lisp")
   (:cl-pamphlet "chunkname1" :pathname "FILE2.lisp")
   (:cl-pamphlet "FILE3.lisp")
   (:cl-pamphlet "FILE4:chunkname2"
 :pathname "FILE4.lisp" :chunk "chunkname2")))
```

2.2 Chunk Centric Definitions

In the case of a large pamphlet file containing many system components it becomes more convenient to address the chunk itself as opposed to focusing on the file - file information will be common to a large number of system components. Repetitive specification of the pamphlet file name is a hint that another style should be used in this case- module level file name specification. In this scheme, the pamphlet is an ASDF module and select chunks within that file are components in the module. Because module names must also be unique within a defsystem, all chunks intended to be used in a single defsystem must be specfied within the module corresponding to their parent pamphlet.Or, if a chunk must be called from the same file later in the system definition, it must use the previous convention for chunk definition - it is not possible to

[1]While this name can be deduced from the pathname and chunk arguments, ASDF's components mechanism is based on CLOS. When CLOS makes a component it needs to have as a supplied argument the name of the component. Rather than re-define the ASDF machinery to provide a deduced name, it is simpler to specify the arguments in the (relatively rare) case where a specified chunk name is not unique within a defsystem. This also retains flexibility in cases where the file:chunkname convention would not produce convenient component names, but the template file-chunkname is related to the naming of intermediate files and for informational purposes when debugging constitutes a "best practic" naming convention.

again define a module with the same pamphlet name.[2] The other main constraint is that a system definition must contain only chunks of one language, otherwise operations on those systems will be greatly complicated.

A system definition in this scheme would take the form (as an example Lisp pamphlets are assumed):

```
(defsystem SYSTEM
  :class literate-system
  :components
  ((asdf-pamphlet "FILE1"
  :components
  ((cl-chunk "chunkname1")
   (cl-chunk "chunkname2")
   (cl-chunk "chunkname3")))
   (asdf-pamphlet "FILE2"
  :components
  ((cl-chunk "chunkname4")
   (cl-chunk "chunkname5")
   (cl-chunk "chunkname6"))))
  (asdf-pamphlet "FILE3"
 :components
 ((cl-chunk "chunkname8")
  (cl-chunk "chunkname9")
  (cl-chunk "chunkname10"))))
```

2.3 Hybrid System Definitions

These two approaches are not mutually exclusive even in a single defsystem definition. While in the above case the style mixing was to allow specification of a chunk from a previously defined pamphlet module out of the module, it may also prove to be useful in projects that contain a mix of large files and small files.

```
(defsystem SYSTEM
  :class literate-system
  :components
  ((asdf-pamphlet "FILE1"
  :components
  ((cl-chunk "chunkname1")
   (cl-chunk "chunkname2")
   (cl-chunk "chunkname3")))
   (:cl-pamphlet "FILE2.lisp")
   (:cl-pamphlet "FILE3:chunkname4"
 :pathname "FILE3.lisp" :chunk "chunkname4"))
  (asdf-pamphlet "FILE4"
 :components
 ((cl-chunk "chunkname5")
  (cl-chunk "chunkname6")
  (cl-chunk "chunkname7"))))
```

3 File Naming and Placement

Unlike a normal Lisp compile, which has only a Lisp source file and a compiled output file, processing a pamphlet to the final state of compiled code and typeset document involves the generation of many intermediate files. Where these files are placed in the file system, and in many cases what they are named, depends on user configurable options.

[2]In such a case using two separate defsystems may prove to be the way to structure the build logic.

It is not ASDF-Literate's job to specify these locations - that is the job of the program or programs making use of the ASDF-Literate functionality - but it does need to provide slots to hold those settings. Also there are situations where it is desirable for multiple defined systems to make use of a common setting for one or more of the output destinations. Consequently, there are five points where this information is potentially retrieved for each step of the pamphlet tangle-and-weave process. In order of decreasing priority:

1. Component level specification of the target directory in question.
2. Module level specification of the target directory in question.
3. System level specification of the target directory in question.
4. A Global (in the `asdf` package) variable containing a meta-system default value.
5. No specification at either system or global level.

Of necessity, the no-specification (nil) case must serve as the default - if no information is supplied the only options are to error out or proceed with a default assumption. The nil case will be dealt with in each perform definition - as a general rule all files created and processed will be in the same directory as the source pamphlet file. (This is the only location that can be guaranteed to be available in all cases.) A System, Module, or Component definition always overrides a global setting, but a global setting will override the nil case.[3]

3.1 Setting up Global Variables

For maximum flexibility, each operation defined for pamphlets or chunks that produces an output file will have its own output path defined. In some cases, these directory names may be the same and different variables may contain the same pathname. This is not a problem, but the design decision for individual variables for each input and destination point ensures maximum flexibility. For input directories, the proper technique is to get the output directory from the previous step (or, in the root operations, to get the source pamphlet from asdf).

A consequence of this decision is that the literate-settings class defined later will need to be updated each time a new operation is added to ASDF-Literate's general capabilities in order to make sure local specification of the directory setting in question is possible. There are some defaults already included for standard literate documentation practices, and they will serve as a model on how to add more if needed. Another option is to create specific specialized classes for other types of work, if the operations to be performed are not generally applicable to all pamphlet types. asdf-pamphlet and asdf-chunk are both specializations of the literate-settings class, and there is no requirement that they are unique.

In the case of globals, they initialized by ASDF-Literate as nil. They are exported from the ASDF package - this allows them to be defined by setf from external packages - e.g. `(setf asdf-literate:tangle-dir (make-pathname :directory "temp"))`. It should be noted that these variables are always expected to contain Lisp pathnames rather than strings - the `make-pathname` command is able to produce any desired pathname.[4] [5]

⟨Standard Settings $_3$⟩ ≡

```
(defvar *tangle-dir*   nil)
(defvar *weave-dir*    nil)
(defvar *compile-dir*  nil)
(defvar *document-dir* nil)
```

Included by 55.

[3] The installed system with its lisp core file needs to define its path structure relative to the location of the core file being used to start the system. - FIGURE OUT HOW TO DO THIS.

[4] For example, the command `(make-pathname :directory '(:relative "src/asdf-literate"))` would result in the relative pathname `#P"src/asdf-literate/"` when evaluated.

[5] `make-pathname` forms assigned to components in defsystem definitions will remain unevaluated in assignment, but this is handled by a call to `eval` in `get-directory-val`.

3.2 Priority of Settings

In order to ensure that local settings correctly override global settings, a special command called `get-directory-val` is defined. The steps are as follows:

1. Check if the component supplied has a non-nil value in the slot corresponding to the dir argument. If it does, return that value.
2. Next, if the slot value is nil and the component has a parent, recursively check the slot value of the parent. The original component is supplied as a third parameter to retain the knowledge of which component information is actually being requested for.
3. If no parent components have a value that is non-nil (up to and including the defsystem definition) return the component-parent-pathname of the original component.

A generic function is used, with a method for all classes of the literate-settings type. This retains flexibility if other behavior is needed for different setups in the future.

⟨Find and Return Directory Settings 4⟩ ≡

```
(defgeneric get-directory-value (literate-obj slot &optional origin))

(defmethod get-directory-value ((obj literate-settings) slot &optional
                                (orig obj))
  (or
   (eval (slot-value obj slot))
   (when (component-parent obj)
     (get-directory-value (component-parent obj) slot orig))
   (component-parent-pathname orig)))
```

Included by 55.

3.3 The Literate-Settings Class

A common set of atributes are shared across the objects used to describe a literate system. The purpose of this special class is to provide slots for key directory variables. For the moment, 4 slots are defined by default to correspond to the default settings defined earlier: tangle-dir, compile-dir, weave-dir, and document-dir. Others may be added.

The initform for each setting corresponds to the global variable. This results in the value of the global variable *current at the time a make-instance is called for a new object* will be taken as the value of the slot for that particular object, and will remain set to its original initialization value even if the global value is later changed. In other words, there is no connection to the global variable itself once an instance of the class is created - *only* during the initial creation process is it referenced.

⟨Defsystem Class for Literate Object Settings 5⟩ ≡

```
(defclass literate-settings ()
  ((tangle-dir
    :initarg :tangle-dir
    :accessor tangle-dir
    :initform *tangle-dir*)
   (compile-dir
    :initarg :compile-dir
    :accessor compile-dir
    :initform *compile-dir*)
   (weave-dir
    :initarg :weave-dir
    :accessor weave-dir
    :initform *weave-dir*)
   (document-dir
    :initarg :document-dir
    :accessor document-dir
    :initform *document-dir*)))
```

Included by 55.

3.4 The Literate-System Class

In order for the specification of directory settings to be possible at the defsystem level (and also for the get-directory-value function to work properly) it is necessary to define a new system class to be used with literate systems. It is specified in traditional asdf fashion:

```
(defsystem system-name
  :class literate-system
  :components
    etc...
```

⟨ Defsystem Class for Literate Systems 6 ⟩ ≡

```
(defclass literate-system (system literate-settings) ())
```

Included by 55.

4 Basic Setup - Pamphlet Classes

There are two basic classes the need to be defined, and a number of derived classes that will specify specific types of pamphlet files. The first class is quite general, and conceptually matches any source file that has a need to define the directory slots for tangling, weaveing, and other standard operations.

⟨ Template Class for Literate Objects 7 ⟩ ≡

```
(defclass literate-template (source-file literate-settings) ())
```

Included by 55.

The second basic class is the pamphlet-module class, since modules corresponding to pamphlets are also a valid point at which to define these settings.

⟨ Module Class for Pamphlet Files 8 ⟩ ≡

```
(defclass pamphlet-module (module literate-settings) ())
```

Included by 55.

4.1 Pathname Commands for Pamphlet Classes

All pamphlet modules will need to return specific pathname values in order to be useful when operating on chunks. Remember that pamphlet modules themselves are only containers for the chunks, and actual file based operations will proceed on chunks. With that in mind, the component-pathname and component-relative-pathname commands for pamphlet-modules are set up to return the pathname of the parent of the pamphlet-module, which corresponds to the target directory information that will be needed in default directory setting situations.

⟨ Pamphlet Module component-pathname Definition 9 ⟩ ≡

```
(defmethod component-pathname ((component pamphlet-module))
  (component-pathname (component-parent component)))
```

Included by 55.

⟨ Pamphlet Module component-relative-pathname Definition 10 ⟩ ≡

```
(defmethod component-relative-pathname ((component pamphlet-module))
  (component-relative-pathname (component-parent component)))
```

Included by 55.

4.2 General Literate Classes

The atomic unit of the ASDF system is a `component` - normally corresponding to a single file. In the case of pamphlet files, this is not strictly true - a `component` may correspond to a singe chunk WITHIN a pamphlet file. So the first order of business is to define a specialized component that knows about pamphlet structure. The option ":chunk" will be used to provide an initial option, "chunk" will be used to access the chunk slot, and an initial value of "*" (the standard top level chunk in a file) will be used if no chunk option is supplied. Since a pamphlet is a source file, the literate-template class is used as a basis for this definition.

⟨ Source-File Class For Pamphlet Files 11 ⟩ ≡

```
(defclass asdf-pamphlet (literate-template)
  ((chunk :initarg :chunk :accessor chunk :initform nil)))
```

Included by 55.

In cases where the definition is chunk-centric, the chunk itself will become a component. Strictly speaking, in this model the chunk is *part* of a source file. However, the literate-template class is still a valid foundation.

⟨ Source-File Class For Code Chunks 12 ⟩ ≡

```
(defclass asdf-chunk (literate-template) ())
```

Included by 55.

4.3 Pathname Commands for asdf-chunk Classes

Just as pamphlet-module specific pathname commands are needed, similar definitions are now needed for chunks. For the chunk objects, we need to define a method that returns the pathname information contained in the parent pamphlet module.

⟨ Code Chunk component-pathname Definition 13 ⟩ ≡

```
(defmethod component-pathname ((asdf-chunk asdf-chunk))
  (make-pathname :name (component-name (component-parent asdf-chunk))
                 :type (source-file-type asdf-chunk
                                         (component-parent asdf-chunk))
                 :defaults (component-pathname
                            (component-parent asdf-chunk))))
```

Included by 55.

⟨ Code Chunk component-relative-pathname Definition 14 ⟩ ≡

```
(defmethod component-relative-pathname ((asdf-chunk asdf-chunk))
  (component-relative-pathname (component-parent asdf-chunk)))
```

Included by 55.

4.4 Language Specific Literate Classes

Now that we have the pamphlet component, a number of convenient special pamphlet types can be defined. No language specific logic is needed in these definitions - they will serve to identify to asdf how to allocate perform methods to each type of pamphlet.

⟨ Source-File Class For Language Specific Pamphlet Files 15 ⟩ ≡

```
(defclass cl-pamphlet (asdf-pamphlet) ())
(defclass boot-pamphlet (asdf-pamphlet) ())
(defclass spad-pamphlet (asdf-pamphlet) ())
```

Included by 55.

Just as there are language specific pamphlets, there are language specific chunks.

⟨ Source-File Class For Language Specific Code Chunks 16 ⟩ ≡

```
(defclass cl-chunk (asdf-chunk) ())
(defclass spad-chunk (asdf-chunk) ())
```

Included by 55.

`source-file-type` is used to specifiy particular types of components and return a string normally associated with the file type.[6]

⟨ Types for Source File Classes 17 ⟩ ≡

```
(defmethod source-file-type ((p asdf-pamphlet) (m module)) "pamphlet")
(defmethod source-file-type ((p pamphlet-module) (m module)) "pamphlet")
(defmethod source-file-type ((p asdf-chunk) (m module)) "pamphlet")
```

Included by 55.

By default ASDF doesn't provide an `output-file-type` method, but in the case of literate files the source file type and the destination file type are almost always different. For generic cases it cannot be defined, but in language specific cases it is useful.

⟨ Generic output-file-type Function 18 ⟩ ≡

```
(defgeneric output-file-type (component))
```

Included by 55.

[6]This will require that the name argument given to the component definition contain not just the root file name but everything up until the pamphlet extension - e.g. "foo.lisp.pamphlet" must be specified as "foo.lisp" rather than "foo" alone. It might be possible to define a system to deal with the code types as well but because pamphlet files may contain multiple types of code it is simpler and cleaner to specify the language via pamphlet type when a particular component is defined.

⟨Language Specific output-file-type Definitions 19⟩ ≡

```
(defmethod output-file-type ((c cl-pamphlet)) "lisp")
(defmethod output-file-type ((c cl-chunk)) "lisp")
(defmethod output-file-type ((c boot-pamphlet)) "boot")
(defmethod output-file-type ((c spad-pamphlet)) "spad")
```

Included by 55.

4.5 Automatic Generation of Output File Names

In order to abstract the handling of intermediate files it is necessary to assign them names automatically based on the source pamphlet and chunk in question. This is not as trivial as it first sounds.

The simplest approach is to simply use the pattern [pamphlet].type for all chunk extractions of the root "*" node, and [chunk] for all non-root nodes. [7] This works best for situations where the chunk is already named in a convenient way, but an isolated chunk name may not be unique throughout a defsystem and result in filename collisions. Also, this approach works ONLY if all chunk names intended to be used as file names are also legal and reasonable on the target build platform. This is not always a safe assumption, particularly in cases where LaTeX markup is used in the chunk name.

A second approach would be to simply assign random strings to chunk names with the gensym command and use those instead of the original chunkname string. This would most likely function, but could make relating files back to their parent pamphlet more difficult in debuggin situations. However, the gensym command's guarantee of uniqueness may not extend to the printed form of the symbol which would be used for filenames. Also, a re-build to update files would lose the original random assignments unless specifically saved and checked for, further increasing the complexity.

A combination of these ideas suggested by Steven Wilson uses the MD5 hash algorithm instead of the gensym command to replace chunk name strings with something acceptable as a file name. This has several advantages. There is a very high degree of probability that the generated hash will constitute a unique name within the pamphlet, and as it is semi-uniquely[8] related to the original chunk name knowing the chunk name will always be enough to find the files corresponding to it. There are several native libraries in Lisp that can produce an md5 sum. While the Ironclad library provides a more general library of such tools, only the md5 routine is needed here; cl-md5 will be used.[9]

Since the human eye will not be able to easily correlate hash names to original chunk names, a prefix will be appended to the result in order to narrow down the possibilities. In the one component per pamphlet scheme this will uniquely identify the output, and in more elaborate schemes it will at least identify the output file's connection to its parent pamphlet. Rather than use the full pamphlet name, the name up to the first "." in the file will be used - henceforth referred to as the pamphlet root name.[10]

⟨Function to Find Root Filename 20⟩ ≡

```
(defun rootname (file)
  (subseq file 0 (position #\. file)))
```

Included by 55.

Also needed in the case of cl-md5 is a routine to convert a Lisp hash object into a string that will work for file output. This approach uses the format command to get the hex representation. [11]

[7]Alternately, to preserve file related information [pamphlet]-[chunk] may be used - from a practical point of view the difference is minor. The pamphlet name is always assumed to be a valid file name, so the difficult still arises from the chunk portion of the name.

[8]Collisions have been found by research teams for the MD5 algorithm, but in this use case no attempt is being made to falsify a chunk name with the same hash - avoiding accidental collisions is still the overwhelming probability of the hash for this situation and that is sufficient. So while the hash is not TRULY uniquely related to one theoretical chunk name, for practical purposes it may be so treated in this application.

[9]The key functionality is to generate an md5sum hash - in Ironclad the job of both hash-to-string and md5 sequence can be done with the code `(byte-array-to-hex-string (digest-sequence :md5 (ascii-string-to-byte-array chunkname)))` in case the programmer wants to switch packages.

[10]Later on, debugging tools will be defined that will take an intermediate file name and identify the chunk it came from automatically by checking the hash against each hash in the pamphlet file until the match is found.

[11]Thanks to chandler on freenode's lisp channel for help with this.

⟨ Function to Convert MD5 Hash to String 21 ⟩ ≡

```
(defun hash-to-string (md5hash)
  (format nil "~{~2,'0X~}" (coerce md5hash 'list)))
```

Included by 55.

Now the actual name generating function can be defined. Because this naming scheme is general, and will be used for tangled files of all types, the function that performs the actual operation needs to take as arguments the pamphlet filename, the chunk name, and the type of file to generate the name for. The actual usage of this information in the code will be in the form not of a string but a Lisp pathname object, so the results of the concatenate are converted with make-pathname.[12]

⟨ Funciton to Generate Intermediate Filenames 22 ⟩ ≡

```
(defun intermediate-name (file chunkname type)
  (make-pathname :name
                 (concatenate 'string (rootname file) "-"
                              (hash-to-string
                               (md5:md5sum-sequence chunkname)))
                 :type type))
```

Included by 55.

So as an example, the command:

```
(intermediate-name "book-vol1.pamphlet" "*" "lisp")
```

would produce the intermediate filename:[13]

```
#P"book-vol1-3389DAE361AF79B04C9C8E7057F60CC6.lisp"
```

5 Chunk Name Extraction from Component Definitions

Since there are multiple methods needed to extract a chunk name based on the situation, a defmethod will be used to handle the chunk query.

⟨ Generic get-chunkname Function 23 ⟩ ≡

```
(defgeneric get-chunkname (c))
```

Included by 55.

5.1 Chunks as Top Level Components

If the component with chunk information is defined as a pamphlet, either the "*" chunk is in use, the component name is the chunk name, or the chunk name has been explicitly specified. These cases are identified as follows:

1. If the component pathname is different from the component name and the chunk slot is non-nil, (chunk c) returns the chunk name.
2. If the component pathname is different from the component name and the chunk slot is nil, (component-name c) returns the chunk name.
3. If the component pathname is not different from the component name, the "*" chunk is in use and "*" should be returned.

[12]As a coding style hint, compile-file-pathname can be used in situations where the binary file extension associated with a particular lisp is needed for the type argument - this allows the code to avoid hard coding knowledge of those extensions and adapt automatically to new lisp environments.
[13]Technically speaking, the result is a Lisp pathname rather than a simple string.

⟨ get-chunkname Method for Pamphlet Files $_{24}$ ⟩ ≡

```
(defmethod get-chunkname ((c asdf-pamphlet))
  (let ((namepath (make-pathname :name (component-name c)
                                 :defaults (component-pathname c))))
    (cond
      ((and (not (equal namepath (component-pathname c)))
            (not (eq (chunk c) nil)))
       (chunk c))
      ((and (not (equal namepath (component-pathname c)))
            (eq (chunk c) nil))
       (component-name c))
      (t "*"))))
```

Included by 55.

5.2 Chunks in Modules

For chunks defined in pamphlet modules, the component name is always the chunk name.

⟨ get-chunkname Method for Code Chunks $_{25}$ ⟩ ≡

```
(defmethod get-chunkname ((c asdf-chunk))
  (component-name c))
```

Included by 55.

6 Operations

In the case of pamphlets, there are seven operations which may be performed:

- Tangle - extract the source code from the pamphlets.
- Compile - compile the extracted source code.
- Load - load the compiled system.
- Load-source - load the uncompiled extracted source code. Primarily for debugging situations.
- Weave - generate a valid LaTeX document from the pamphlet.
- Document - generate the final form (dvi/pdf) from the generated LaTeX.
- Build - Compile the system and Document it.

Compile, load, and load-source are already defined in ASDF as operations - it remains only to provide specialized methods for dealing with literate objects. Tangle, weave, document and build are new - they are specific to literate processing and must be defined as operations:

⟨ Classes for Literate Programming Operations $_{26}$ ⟩ ≡

```
(defclass info-op (operation) ())
(defclass tangle-op (operation) ())
(defclass weave-op (operation) ())
(defclass document-op (operation) ())
(defclass build-op (operation) ())
```

Included by 55.

7 Perform - Pamphlets as Components

Perform methods are where the core logic of the operations on files is defined - it is here the necessary incantations to translate pamphlet files into running code are defined.

ASDF performs `operations` on systems using perform methods.

Rather than define all of the logic in a single method, it is customary to treat each independent step as its own method and use dependency rules to ensure methods are invoked when needed. This allows additional flexibility, since lower level operations may be performed for debugging without forcing the developer to complete all steps. Say, for example, the developer wishes to test the tangle operation on a system but not compile it or load it. They can then operate on the system with the tangle operation only, using the same system definition used for the full compile.

Next, the tangle perform methods themselves are defined. Because each type must be handled differently, the methods are specialized on the asdf-pamphlet and asdf-chunk classes.

At this point, the tangle operation is agnostic to the particular language contained in the chunk - as a result, it is possible to make a very general defintion for all cases.

⟨Perform Method for Tangling Pamphlets 27⟩ ≡

```
(defmethod perform ((operation tangle-op) (c asdf-pamphlet))
  (let* ((tangle-op (make-instance 'tangle-op))
         (in-file (first (input-files tangle-op c)))
         (out-file (first (output-files tangle-op c))))
    (format t "~&Tangling ~A to ~A~%~%" in-file out-file)
    (tangle in-file out-file (get-chunkname c))))
```

Included by 55.

An output-files command is needed to generate the target path name,

⟨Output Files Method for Pamphlet Tangling Operation 28⟩ ≡

```
(defmethod output-files ((operation tangle-op) (c asdf-pamphlet))
  (list (merge-pathnames (get-directory-value c 'tangle-dir)
                         (intermediate-name
                          (file-namestring
                           (component-relative-pathname c))
                          (get-chunkname c) (output-file-type c)))))
```

Included by 55.

Code chunks also need a tangle method:

⟨Perform Method for Tangling Code Chunks 29⟩ ≡

```
(defmethod perform ((operation tangle-op) (c asdf-chunk))
  (let* ((tangle-op (make-instance 'tangle-op))
         (in-file (first (input-files tangle-op c)))
         (out-file (first (output-files tangle-op c))))
    (format t "~&Tangling ~A to ~a~%" in-file out-file)
    (tangle in-file out-file (get-chunkname c))))
```

Included by 55.

and a corresponding output-files method:

⟨Output Files Method for Code Chunk Tangling Operation 30⟩ ≡

```
(defmethod output-files ((operation tangle-op) (c cl-chunk))
  (list (merge-pathnames (get-directory-value c 'tangle-dir)
                         (intermediate-name
                          (component-name (component-parent c))
                          (get-chunkname c) (output-file-type c)))))
```

Included by 55.

The weave operation is also agnostic to the particular language contained in the chunks, but there is an interesting wrinkle in that many chunk definitions will trigger the same weave step. The solution is ASDF's operation-done-p. It will report to ASDF that a given pamphlet has already been woven, and thus avoide unnecessary reprocessing.

It should be noted that unlike chunk based files, weave and LaTeX documents will use only the root name of the pamphlet file - they are not specific to any one chunk.

⟨Perform Method for Weaving Pamphlets 31⟩ ≡

```
(defmethod perform ((operation weave-op) (c asdf-pamphlet))
  (let* ((weave-op (make-instance 'weave-op))
         (in-file (car (input-files (make-instance 'weave-op) c)))
         (out-file (car (output-files (make-instance 'weave-op) c))))
    (format t "~&Weaving ~A to ~A~%" in-file out-file)
    (weave in-file out-file)))
```

Included by 55.

An output-files command is needed to generate the target path name,

⟨Output Files Method for Pamphlet Weaving Operation 32⟩ ≡

```
(defmethod output-files ((operation weave-op) (c asdf-pamphlet))
  (let ((rootname (rootname (file-namestring
                             (component-relative-pathname c)))))
    (list (merge-pathnames (get-directory-value c 'weave-dir)
                           (make-pathname :name rootname
                                          :type "tex")))))
```

Included by 55.

In the case of weaving chunks, matters are a bit different - rather than weaving the individual source code of the chunk in question, the task is to ensure the parent file has been woven.

⟨Perform Method for Weaving Code Chunks 33⟩ ≡

```
(defmethod perform ((operation weave-op) (c asdf-chunk))
  (let* ((weave-op (make-instance 'weave-op))
         (in-file (first (input-files (make-instance 'weave-op) c)))
         (out-file (first (output-files (make-instance 'weave-op) c))))
    (format t "~&Weaving pamphlet ~A which is parent of chunk ~A to ~A~%"
            in-file  c out-file)
    (weave in-file out-file)))
```

Included by 55.

⟨Output Files Method for Code Chunk Weaving Operation 34⟩ ≡

```
(defmethod output-files ((operation weave-op) (c cl-chunk))
  (let ((rootname (rootname (file-namestring (component-name
                                              (component-parent c))))))
    (list (merge-pathnames (get-directory-value c 'weave-dir)
                           (make-pathname :name rootname :type "tex")))))
```

Included by 55.

7.1 Compile

Compiling, in the case of Lisp files, is as simple as invoking compile-file on the correct input files and specifying the correct output targets. The following implements a portable method to obtain the correct file type extension used by the lisp compiler.

⟨Fucntion to Find Compiled File Extension 35⟩ ≡

```
(defun default-compile-extension ()
  (pathname-type (compile-file-pathname "a.lisp")))
```

Included by 55.

Once that is available, perform methods, input-files, and output-files can be defined for cl-pamphlets.

⟨Perform Method for Compiling Lisp Pamphlets 36⟩ ≡

```
(defmethod perform ((operation compile-op) (c cl-pamphlet))
  (let* ((compile-op (make-instance 'compile-op))
         (in-file (first (input-files compile-op c)))
         (out-file (first (output-files compile-op c))))
    (format t "~&Compiling component ~A from ~A to ~A~%"
            c in-file out-file)
    (compile-file in-file :output-file out-file)))
```

Included by 55.

⟨Input Files Method for Lisp Pamphlet Compiling Operation 37⟩ ≡

```
(defmethod input-files ((operation compile-op) (c cl-pamphlet))
  (output-files (make-instance 'tangle-op) c))
```

Included by 55.

⟨Output Files Method for Lisp Pamphlet Compiling Operation 38⟩ ≡

```
(defmethod output-files ((operation compile-op) (c cl-pamphlet))
  (list (merge-pathnames (get-directory-value c 'compile-dir)
                         (intermediate-name
                          (file-namestring
                           (component-relative-pathname c))
                          (get-chunkname c)
                          (default-compile-extension)))))
```

Included by 55.

cl-chunks work basically the same way - the logic making them different is hidden behind the various function calls.

⟨Perform Method for Compiling Lisp Code Chunks 39⟩ ≡

```
(defmethod perform ((operation compile-op) (c cl-chunk))
  (let* ((compile-op (make-instance 'compile-op))
         (in-file (first (input-files compile-op c)))
         (out-file (first (output-files compile-op c))))
    (format t "Compiling component ~A from ~A to ~A~%~%"
            c in-file out-file)
    (compile-file in-file :output-file out-file)))
```

Included by 55.

⟨Input Files Method for Lisp Code Chunk Compiling Operation 40⟩ ≡

```
(defmethod input-files ((operation compile-op) (c cl-chunk))
  (output-files (make-instance 'tangle-op) c))
```

Included by 55.

⟨ Output Files Method for Lisp Code Chunk Compiling Operation 41 ⟩ ≡

```
(defmethod output-files ((operation compile-op) (c cl-chunk))
  (list (merge-pathnames (get-directory-value c 'compile-dir)
                         (intermediate-name
                          (component-name (component-parent c))
                          (get-chunkname c)
                          (default-compile-extension)))))
```

Included by 55.

A successful compile operation depends on a successful tangle operation, so this information is supplied to ASDF:

⟨ component-depends-on Methods for Compilation 42 ⟩ ≡

```
(defmethod component-depends-on ((operation compile-op) (c cl-pamphlet))
  (cons (list 'tangle-op (component-name c))
        (call-next-method)))

(defmethod component-depends-on ((operation compile-op) (c cl-chunk))
  (cons (list 'tangle-op (component-name c))
        (call-next-method)))
```

Included by 55.

7.2 Load

ASDF's normal load method will suffice here - the only thing needed is to specify this and inform ASDF what files to load. A load operation of this type is intended to load compiled files, so it depends on a successful compile operation.

⟨ Methods for Loading Compiled Lisp Pamphlets 43 ⟩ ≡

```
(defmethod perform ((o load-op) (c cl-pamphlet))
  (mapcar #'load (input-files o c)))

(defmethod input-files ((operation load-op) (c cl-pamphlet))
  (output-files (make-instance 'compile-op) c))

(defmethod component-depends-on ((operation load-op) (c cl-pamphlet))
  (cons (list 'compile-op (component-name c))
        (call-next-method)))
```

Included by 55.

⟨ Methods for Loading Compiled Lisp Code Chunks 44 ⟩ ≡

```
(defmethod perform ((o load-op) (c cl-chunk))
  (mapcar #'load (input-files o c)))

(defmethod input-files ((operation load-op) (c cl-chunk))
  (output-files (make-instance 'compile-op) c))

(defmethod component-depends-on ((operation load-op) (c cl-chunk))
  (cons (list 'compile-op (component-name c))
        (call-next-method)))
```

Included by 55.

7.3 Load-Source

This is an option to skip the compile step and load directly from the lisp source files. Normally it is used only in debugging, as the compiled file speed will be greater.

⟨Methods for Loading Lisp Source from Lisp Pamphlets 45⟩ ≡

```
(defmethod perform ((o load-source-op) (c cl-pamphlet))
  (mapcar #'load (input-files o c)))

(defmethod input-files ((operation load-source-op) (c cl-pamphlet))
  (output-files (make-instance 'tangle-op) c))

(defmethod component-depends-on ((operation load-source-op)
                                 (c cl-pamphlet))
  (cons (list 'tangle-op (component-name c))
        (call-next-method)))
```

Included by 55.

⟨Methods for Loading Lisp Source from Lisp Code Chunks 46⟩ ≡

```
(defmethod perform ((o load-source-op) (c cl-chunk))
  (mapcar #'load (input-files o c)))

(defmethod input-files ((operation load-source-op) (c cl-chunk))
  (output-files (make-instance 'tangle-op) c))

(defmethod component-depends-on ((operation load-source-op) (c cl-chunk))
  (cons (list 'tangle-op (component-name c))
        (call-next-method)))
```

Included by 55.

7.4 Document

LaTeX, like weave and tangle, does not have language dependent behavior. Thus, it is possible to define very general operations. Those familiar with LaTeXare doubtless aware of the large number of intermediate files that are generated in the process - the responsibility for cleanup is given to the tool providing access to the LaTeXcommand.

⟨Perform Method for LaTeX Processing of Pamphlets 47⟩ ≡

```
(defmethod perform ((operation document-op) (c asdf-pamphlet))
  (let* ((document-op (make-instance 'document-op))
         (in-file (first (input-files document-op c)))
         (out-file (first (output-files document-op c))))
    (format t "LaTeXing pamphlet ~A to ~A~%~%" in-file out-file)
    (latex in-file out-file)))
```

Included by 55.

The input files for this operation are the output of the weave operation:

⟨Input Files Method for Pamphlet Documenting Operation 48⟩ ≡

```
(defmethod input-files ((operation document-op) (c asdf-pamphlet))
  (output-files (make-instance 'weave-op) c))
```

Included by 55.

An output-files command is needed to generate the target path name,

⟨Output Files Method for Pamphlet Documenting Operation 49⟩ ≡

```
(defmethod output-files ((operation document-op) (c asdf-pamphlet))
  (let ((rootname (rootname (file-namestring (component-relative-pathname c)))))
    (list (merge-pathnames (get-directory-value c 'document-dir)
                           (make-pathname :name rootname :type "dvi")))))
```

Included by 55.

⟨Perform Method for LaTeX Processing of Code Chunks 50⟩ ≡

```
(defmethod perform ((operation document-op) (c asdf-chunk))
  (let* ((document-op (make-instance 'document-op))
         (in-file (first (input-files document-op c)))
         (out-file (first (output-files document-op c))))
    (format t "LaTeXing pamphlet ~A to ~A~%~%" in-file out-file)
    (latex in-file out-file)))
```

Included by 55.

⟨Input Files Method for Code Chunk Documenting Operation 51⟩ ≡

```
(defmethod input-files ((operation document-op) (c asdf-chunk))
  (output-files (make-instance 'weave-op) c))
```

Included by 55.

⟨Output Files Method for Code Chunk Documenting Operation 52⟩ ≡

```
(defmethod output-files ((operation document-op) (c cl-chunk))
  (let ((rootname (rootname (file-namestring (component-name
                                              (component-parent c))))))
    (list (merge-pathnames (get-directory-value c 'document-dir)
                           (make-pathname :name rootname :type "dvi")))))
```

Included by 55.

Just as the compile operation needs a successful tangle, the document operation needs a successful weave:

⟨component-depends-on Methods for Documentation 53⟩ ≡

```
(defmethod component-depends-on ((operation document-op) (c cl-pamphlet))
  (cons (list 'weave-op (component-name c))
        (call-next-method)))

(defmethod component-depends-on ((operation document-op) (c cl-chunk))
  (cons (list 'weave-op (component-name c))
        (call-next-method)))
```

Included by 55.

7.5 Build

The build operation is an operation for performing both the code and documentation operations on a system.

⟨Perform Method for Compiling and Documenting 54⟩ ≡

```
(defmethod perform ((operation build-op) (c component))
  (operate 'load-op c)
  (operate 'document-op c))
```

Included by 55.

8 Completed Operations

ASDF-Literate is defined in its own package, but must use a great deal of ASDF functionality. Fortunately, only one function (`component-parent-pathname`) is both used in asdf-literate and unexported in asdf - rather than change asdf, its definition is reproduced here.

⟨* 55⟩ ≡

```
⟨Copyright 1⟩
⟨License 2⟩
(defpackage "ASDF-LITERATE"
  (:nicknames "ASDF-LIT")
  (:use :cl :asdf :md5 :cl-exttex :axweb)
  (:export "*TANGLE-DIR*"
           "*WEAVE-DIR*"
           "*COMPILE-DIR*"
           "*DOCUMENT-DIR*"
           "TANGLE-OP"
           "WEAVE-OP"
           "DOCUMENT-OP"
           "BUILD-OP"
           "LITERATE-SYSTEM"
           "CL-PAMPHLET"))

(in-package "ASDF-LITERATE")

(defun component-parent-pathname (component)
  (let ((parent (component-parent component)))
    (if (null parent)
        *default-pathname-defaults*
        (component-pathname parent))))

⟨Standard Settings 3⟩
⟨Defsystem Class for Literate Object Settings 5⟩
⟨Defsystem Class for Literate Systems 6⟩
⟨Template Class for Literate Objects 7⟩
⟨Module Class for Pamphlet Files 8⟩
⟨Pamphlet Module component-pathname Definition 9⟩
⟨Pamphlet Module component-relative-pathname Definition 10⟩
⟨Source-File Class For Pamphlet Files 11⟩
⟨Source-File Class For Code Chunks 12⟩
⟨Code Chunk component-pathname Definition 13⟩
⟨Code Chunk component-relative-pathname Definition 14⟩
⟨Source-File Class For Language Specific Pamphlet Files 15⟩
⟨Source-File Class For Language Specific Code Chunks 16⟩
⟨Types for Source File Classes 17⟩
⟨Generic output-file-type Function 18⟩
⟨Language Specific output-file-type Definitions 19⟩
⟨Generic get-chunkname Function 23⟩
⟨get-chunkname Method for Pamphlet Files 24⟩
```

⟨get-chunkname Method for Code Chunks 25⟩
⟨Function to Find Root Filename 20⟩
⟨Function to Convert MD5 Hash to String 21⟩
⟨Funciton to Generate Intermediate Filenames 22⟩
⟨Find and Return Directory Settings 4⟩
⟨Classes for Literate Programming Operations 26⟩
⟨Perform Method for Tangling Pamphlets 27⟩
⟨Output Files Method for Pamphlet Tangling Operation 28⟩
⟨Perform Method for Tangling Code Chunks 29⟩
⟨Output Files Method for Code Chunk Tangling Operation 30⟩
⟨Perform Method for Weaving Pamphlets 31⟩
⟨Output Files Method for Pamphlet Weaving Operation 32⟩
⟨Perform Method for Weaving Code Chunks 33⟩
⟨Output Files Method for Code Chunk Weaving Operation 34⟩
⟨Fucntion to Find Compiled File Extension 35⟩
⟨Perform Method for Compiling Lisp Pamphlets 36⟩
⟨Input Files Method for Lisp Pamphlet Compiling Operation 37⟩
⟨Output Files Method for Lisp Pamphlet Compiling Operation 38⟩
⟨Perform Method for Compiling Lisp Code Chunks 39⟩
⟨Input Files Method for Lisp Code Chunk Compiling Operation 40⟩
⟨Output Files Method for Lisp Code Chunk Compiling Operation 41⟩
⟨component-depends-on Methods for Compilation 42⟩
⟨Methods for Loading Compiled Lisp Pamphlets 43⟩
⟨Methods for Loading Compiled Lisp Code Chunks 44⟩
⟨Methods for Loading Lisp Source from Lisp Pamphlets 45⟩
⟨Methods for Loading Lisp Source from Lisp Code Chunks 46⟩
⟨Perform Method for LaTeX Processing of Pamphlets 47⟩
⟨Input Files Method for Pamphlet Documenting Operation 48⟩
⟨Output Files Method for Pamphlet Documenting Operation 49⟩
⟨Perform Method for LaTeX Processing of Code Chunks 50⟩
⟨Input Files Method for Code Chunk Documenting Operation 51⟩
⟨Output Files Method for Code Chunk Documenting Operation 52⟩
⟨component-depends-on Methods for Documentation 53⟩
⟨Perform Method for Compiling and Documenting 54⟩