Semi-literate programming

John Maraist

SIFT, LLC
Minneapolis, Minnesota, USA

International Lisp Conference 2010
(Lightning talk — revised slides)
Why literate programming?

On Tuesday we heard from Roy Turner about literate programming:

*Hey, your manual and your code say two different things!*
What I more usually hear

Hey, *your manual, your docstrings and your code say three different things!*
Literate programming aims for comprehensive explanation

Semi-literate programming has more modest goals.

- Not worried about explaining the algorithm.
  - We need to document the API.
- Don’t need to generate a whole, complete document.
  - But some snippets to \texttt{\textbackslash input} would be great.
- Include docstrings in what we generate.
  - Emacs, for example, draws from the docstrings.
Example: NST’s `def-fixture` macro

Note: no docstring.
Example: NST’s def-fixture macro

Below the defmacro, we declare its documentation.

- compiler-macro, as opposed to function, variable, etc.
- Different forms of documentation: :latex, :plain
Example: NST’s `def-fixture` macro

`:callspec` gives a user’s view of the lambda list.

- (Can be) more instructive than the verbatim lambda list.
- To do: check against lambda list for compatibility.
Example: NST’s \texttt{def-fixture} macro

\texttt{params} gives documentation of each parameter

- À la javadoc comments.
- Also useful for slots.
- Note mixture of \LaTeX{} and plain text.
1 Fixtures

Fixtures are data structures and values which may be referred to by name during testing. NST provides the ability to use fixtures across multiple tests and test groups, and to inject fixtures into the runtime namespace for debugging. A set of fixtures is defined using the def-fixtures macro:

```lisp
(defun fixture-name (special)
  (...)
  (:name fixture-name)
  (...)
)
```

`fixture-name` The name to be associated with this set of fixtures.

`inner` List of declarations to be made inside the let-binding of names of any use of this fixture. Do not include the “declare” keyword here; NST adds
Example: The generated docstring

```
*common-lisp* fixture.lisp

**nst(5):**\ (documentation 'def-fixtures 'compiler-macro)

"Fixtures are data structures and values which may be referred to by name during testing. NST provides the ability to use fixtures across multiple tests and test groups, and to inject fixtures into the runtime namespace for debugging. A set of fixtures is defined using the def-fixtures macro:

```
(def-fixtures fixture-name ([ :special (NAME ... NAME
   (:fixture NAME ... NAME)) ]
   [:outer FORM ] [:inner FORM ]
   [:setup FORM ] [:cleanup FORM ]
   [:startup FORM ] [:finish FORM ]
   [:documentation STRING ] [:cache FLAG ]
   [:export-names FLAG ]
   [:export-fixture-name FLAG ]
   [:export-bound-names FLAG ])

```

```
([ ([ :cache FLAG ] ] NAME FORM)
 ... ([ [ :cache FLAG ] ] NAME FORM))
```

**fixture-name**
The name to be associated with this set of fixtures.

**inner**
List of declarations to be made inside the let-binding of names of any use
```
ISO8---x---ACL Idle *common-lisp*  (Inferior Common Lisp)---62%---
```
Facts, benefits, problems

- There’s an extensible model behind it
  - Dispatch on package or target type to define system-specific documentation models.
  - Extend the standard documentation models for richer/finer information.
  - Similarly for output models.
  - Early days — especially the API for customization is likely to change.
  - Is it the right document model?

- Wordiness in the code.
  - Quoting all of the backslashes is tedious.

- It’s not where the docstrings are.
  - Shadowing `cl:function et al.`
The `defdoc` library

Problems notwithstanding, it does now work.
- Developed to coordinate documentation for NST.
- Used via ASDF.

Currently part of NST
- Use the svn HEAD:  
- Ironically, very little documentation right now!
- Will separate from NST at some point.