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!

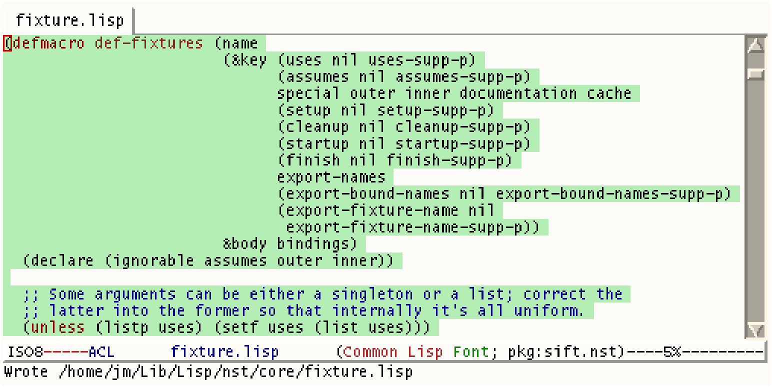
Literate programming aims for comprehensive explanation

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 `\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.



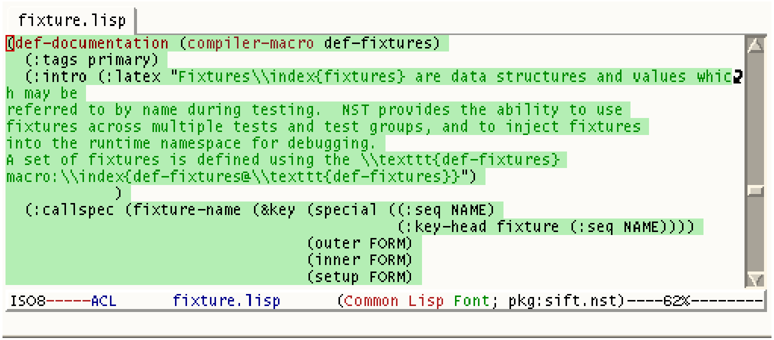
```
fixture.lisp
(defmacro def-fixtures (name
                        (&key (uses nil uses-supp-p)
                              (assumes nil assumes-supp-p)
                              special outer inner documentation cache
                              (setup nil setup-supp-p)
                              (cleanup nil cleanup-supp-p)
                              (startup nil startup-supp-p)
                              (finish nil finish-supp-p)
                              export-names
                              (export-bound-names nil export-bound-names-supp-p)
                              (export-fixture-name nil
                               export-fixture-name-supp-p))
                        &body bindings)
  (declare (ignorable assumes outer inner))

  ;; Some arguments can be either a singleton or a list; correct the
  ;; latter into the former so that internally it's all uniform.
  (unless (listp uses) (setf uses (list uses)))
ISO8-----ACL       fixture.lisp       (Common Lisp Font; pkg:sift.nst)----5%---------
Wrote /home/jm/Lib/Lisp/nst/core/fixture.lisp
```

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`

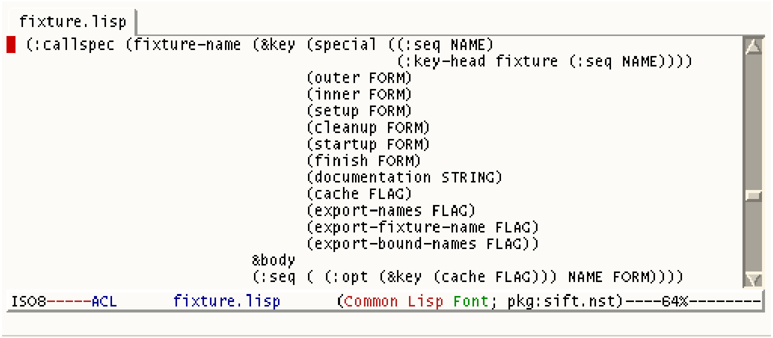


```
fixture.lisp
(def-documentation (compiler-macro def-fixtures)
  (:tags primary)
  (:intro (:latex "Fixtures\\index{fixtures} are data structures and values whic
h may be
referred to by name during testing.  NST provides the ability to use
fixtures across multiple tests and test groups, and to inject fixtures
into the runtime namespace for debugging.
A set of fixtures is defined using the \\texttt{def-fixtures}
macro:\\index{def-fixtures@\\texttt{def-fixtures}}")
            )
  (:callspec (fixture-name (&key (special ((:seq NAME)
                                           (:key-head fixture (:seq NAME)))
                                  (outer FORM)
                                  (inner FORM)
                                  (setup FORM)
ISO8-----ACL      fixture.lisp      (Common Lisp Font; pkg:sift.nst)----62%--------
```

Example: NST's `def-fixture` macro

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

- (Can be) more instructive that the verbatim lambda list.
- To do: check against lambda list for compatibility.



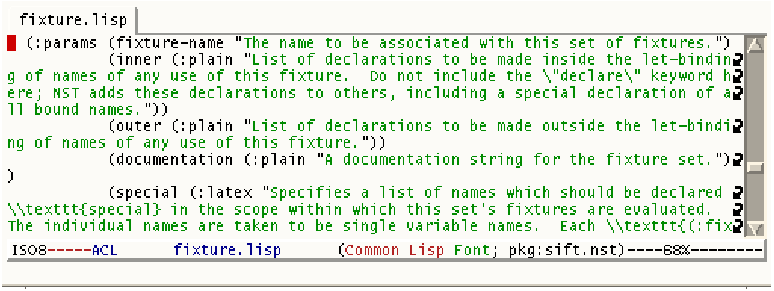
fixture.lisp

```
(:callspec (fixture-name (&key (special ((:seq NAME)
                                          (:key-head fixture (:seq 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))
                          &body
                          (:seq ( (:opt (&key (cache FLAG))) NAME FORM)))
ISO8-----ACL      fixture.lisp      (Common Lisp Font; pkg:sift.nst)----64%--------
```

Example: NST's `def-fixture` macro

`:params` gives documentation of each parameter

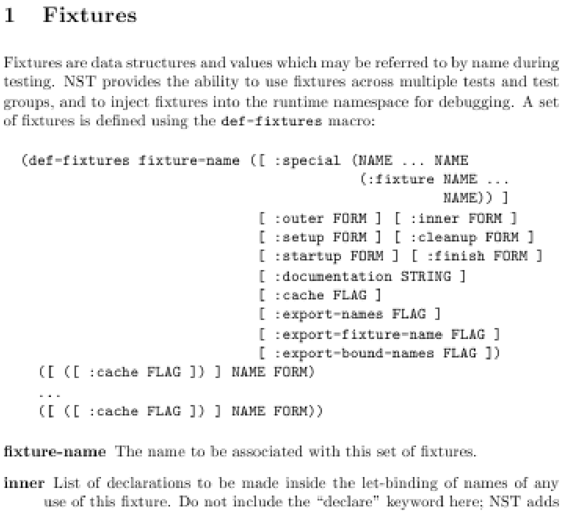
- À la javadoc comments.
- Also useful for slots.
- Note mixture of LaTeX and plain text.



```
fixture.lisp
 (:params (fixture-name "The name to be associated with this set of fixtures.")
          (inner (:plain "List of declarations to be made inside the let-bindin
g of names of any use of this fixture.  Do not include the \"declare\" keyword h
ere; NST adds these declarations to others, including a special declaration of a
ll bound names."))
          (outer (:plain "List of declarations to be made outside the let-bindi
ng of names of any use of this fixture."))
          (documentation (:plain "A documentation string for the fixture set.")
)
          (special (:latex "Specifies a list of names which should be declared 
\\texttt{special} in the scope within which this set's fixtures are evaluated.  
The individual names are taken to be single variable names.  Each \\texttt{(:fix
ISO8-----ACL      fixture.lisp      (Common Lisp Font; pkg:sift.nst)----68%--------
```

SIFT

Example: Generated for the manual



1 Fixtures

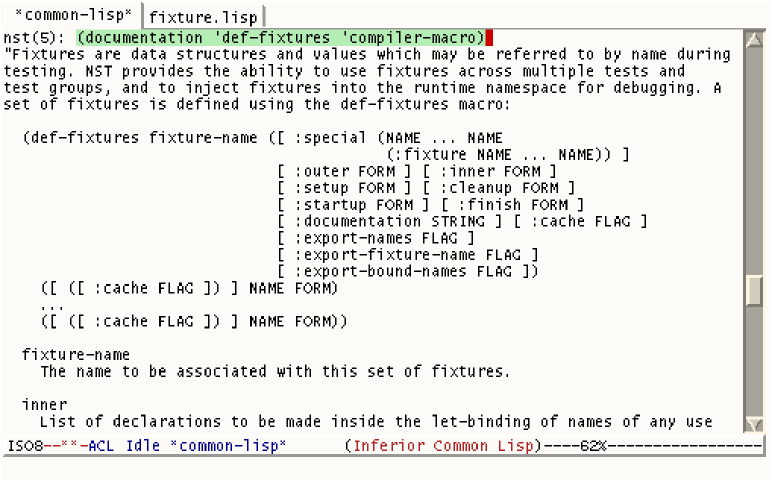
Fixtures are data structures and values which may be referred to by name during testing. NST provides the ability to use fixtures across multiple tests and test groups, and to inject fixtures into the runtime namespace for debugging. A set of fixtures is defined using the **def-fixtures** macro:

```
(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 of this fixture. Do not include the "declare" keyword here; NST adds

Example: The generated docstring



```
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--**-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.
- Generates docstrings, manual, quick-reference sheet.
- Used via ASDF.

Currently part of NST

- Use the svn HEAD:
  https://svn.sift.info:3333/svn/nst/trunk/ext/defdoc.
- Ironically, very little documentation right now!
- Will separate from NST at some point.