Creating New Custom Symbols: new-symbol.lisp

Roy M. Turner

Spring, 2021

Description

This file goes beyond the built-in `gensym` function for creating symbols. In particular, you can create a sequence of symbols whose name starts with whatever prefix you like, as opposed to `gensym`'s limitation of ones that look like `G596`, `G597`, etc. This is done using the `new-symbol` macro defined here. In addition, whereas `gensym` just provides a symbol without *interning* it (i.e, putting it into the current package's symbol table), those produced will be interned automatically.

You may ask: "Why would I need this?" Primarily this is useful for creating new names for objects or other things that have some sort of meaning. For example, if you have a class `robot` to simulate a robot, then instead of you having to come up with a name for each robot you create, you can do it this way:

```
(defclass robot ()
  ((name (new-symbol 'robot))
   ...
   ))
```

The first robot will be named `ROBOT0`, the second `ROBOT1`, and so on.

The `new-symbol` macro takes a single, optional argument specifying the prefix. This can be either a symbol:

```
(new-symbol 'robot)
```

or a string:

```
(new-symbol "ROBOT")
```

Note that if you specify a string and you are using Lisp's default setting, case-insensitive, then you should probably uppercase the string, otherwise you'll get something that looks like this

```
|robot0|
```

which is how Lisp handles lowercase names; you won't be able to type this in easily.

Without an argument, `new-symbol` returns symbols that look like `S1`, `S2`, and so forth.

This file is part of the utilities I'm providing to you, but if you want to use this file by itself, just put this in your code:

```
(load "new-symbol")
```

if the file `new-symbol.lisp` is in the same directory, else replace the name with the path to where it lives.[1]

The code in the file is in the `NEWSYMBOL` package. You can call `new-symbol` from the default package you are likely using (`cl-user`) as:

```
(newsymbol:new-symbol)
```

or by using the package's nickname, `SYM`:

```
(sym:new-symbol)
```

Alternatively, you can import the symbol into your current package by:

```
(import '(sym:new-symbol))
```

after which you can just do:

```
(new-symbol)
```

[1] Both the file `new-generator.lisp` and this documentation are created from the same Emacs Org Mode file; this is an example of *literate programming* (that's the term for it, not necessarily a commentary on the quality of my writing!).

Code

First, define the package and change to that package for the rest of the file:

```
1  (unless (find-package "SYM")
2    (defpackage "NEWSYMBOL"
3      (:use "COMMON-LISP")
4      (:nicknames "SYM")
5      (:export "NEW-SYMBOL")
6      ))
7
8  (in-package sym)
```

A variable is next that defines the default symbol prefix. I use the convention that variable names surrounded by asterisks denote global variables, almost always defined via either `defvar`, `defparameter`, or `setq` at the top level.

As an aside, these are *dynamic* variables, i.e., they are dynamically-scoped (and are often referred to as "special variables"), rather than the usual lexically-scoped variables in Common Lisp. Such variables are very useful in some contexts and extremely confusing in others. While lexical variables behave like the ones you're used to in Python and other languages, a dynamic variable's value, if not local, is looked for in the *run-time stack*, not based on where it is defined in the code. This can be handy. For example, in some of my research code, objects can print to different locations while all using the same print methods;

I do this by having different "message handler" objects that do the printing, then binding a special variable used by the print methods to the one I want for the object being created.

In any case, if you want to change the default symbol prefix, change the value assigned to the variable `*default-symbol-prefix*`. The value of another dynamic variable, `*symbol-generator*`, will be set at the end of the file to an instance of the `symbol-generator` class.

```
9  (defvar *default-symbol-prefix* 's)
```

An instance of the `symbol-generator` class is where all the prefixes are stored, and its methods are what create new symbols. **Note:** No matter how tempted you might be (for whatever reason), never, ever, make two instances of this class or you will get some *truly* bizarre behavior, since the new instance will start over at 0 for the suffixes of all symbols. If you accidentally do this, your best bet is to exit Lisp and restart it.

Note that the symbol prefixes are stored in a hash table for fast access.

```
10  (defclass symbol-generator ()
11    (
12     (symbol-prefixes :initform (make-hash-table))
13     ))
```

When you create an instance of a CLOS object, e.g., with `make-instance`, Lisp calls the method `initialize-instance` to set it up correctly. A common thing to need to do is to change the way an object is initialized. Rather than redefine `initialize-instance`, mechanisms exist in CLOS to specify methods with the same name that are executed before, after, and/or around a method. This is what the next method does: it is an `:after` method that runs to set the default prefix after the instance has been set up by `initialize-instance` itself.

Note the way hash tables are accessed in Lisp. To get a value from it, we use `gethash`. To *set* a value, on the other hand, we typically use the generic setting function,[2] `setf`. This sets the location specified by the first argument to the value of the second. We can use `setf` like we'd use `setq`:

```
(setf a 3)
```

will set the variable `A` to 3.[3]

```
14  (defmethod initialize-instance :after ((self symbol-generator) &rest l)
15    (declare (ignore l))
16    (setf (gethash *default-symbol-prefix* (slot-value self 'symbol-prefixes)) 0))
17
```

[2] Actually, `setf` is a "special form", meaning that although it looks like a function, it doesn't really operate in the same way. In particular, `setf` doesn't so much evaluate its first argument as determine what in memory it refers to.

[3] Okay, *technically* it's not a variable so much as a symbol, and the pedantically-correct term is that we bind the value 3 to the symbol `A`, or that we set the value cell of `A` to 3. But it's pretty close. Looks like a variable, eh?

```
18  (defmethod make-new-symbol ((self symbol-generator) &optional
19       (symbol-prefix *default-symbol-prefix*))
20    (with-slots (symbol-prefixes) self
21      (when (stringp symbol-prefix)
22        (setq symbol-prefix (intern symbol-prefix))) ;convert to symbol
23
24      (unless symbol-prefix
25        (setq symbol-prefix *default-symbol-prefix*))
26
27      (let* ((next-count (or (gethash symbol-prefix symbol-prefixes) 0))
28      (newsym (intern (concatenate 'string (symbol-name symbol-prefix)
29   (princ-to-string next-count)))))
30        (setf (gethash symbol-prefix symbol-prefixes) (1+ next-count))
31        newsym)))
32
```

Create a variable to hold an instance of `symbol-generator` and initialize it. I am using `defvar` rather than `defparameter` here because it isn't re-evaluated should this file be reloaded in a running Lisp; see comment above about why you *really* don't want to create another one of these.

```
33  (defvar *symbol-generator* (make-instance 'symbol-generator))
```

Now create the `new-symbol` macro. I could have made it a function, but since it's so short, no reason to incur the overhead of a function call. This *does* mean that this file needs to be loaded prior to any other source file that uses the macro, though, given how macros work.

And, yes, I could have made it not evaluate its argument, so you could do:

```
(new-symbol foo)
```

but I thought the difference from a normal function call might be more confusing than helpful.

```
34  (defmacro new-symbol (&optional prefix)
35    `(make-new-symbol *symbol-generator* ,prefix))
```

That's all, folks.