Naming and Finding

Peter Norvig (and maybe Fry)

1. Introduction

This paper describes some of the naming conventions for Dylan symbols, and the tools for helping users find existing symbols (and hopefully to find the right functionality behind the symbols).

In general, there are two goals: (1) Make it easy for a novice to start being productive quickly by providing enough menu-based support to eliminate the "blank screen" syndrome. (2) Make it easy for an experienced user to find existing resources, and to know what to name a new object.

2. The Sub-Atomic Physics of Names

Most Lisp systems treat symbols as *atoms*. However, most users treat these atoms as complex names composed of words. That is, **delete-file** is composed of the words **delete** and **file**, while **makunbound** is composed of **make** and **unbound** (where **unbound** is in turn related to **bound**). Obviously, **delete-file** is easier to deal with than **makunbound**. We feel that Dylan will be easier to deal with if names are chosen in a way that obeys some simple guidelines:

- Use English words, not abbreviations.
- Combine words with hyphens, not just concatenation.
- Standard English morphology such as prefixes (the **super** in **superclass**) and suffixes (the **es** in **all-superclasses**) is acceptable.
- Choose a set of words and stick to them. Don't use **make** in some symbols and **create** in others for no good reason. Use the words that Dylan provides. For example, Dylan uses the word **slot**, not "field" or "instance variable."
- Combine words with standard patterns. One common pattern is *verb-object*: **add-method**, **freeze-methods**, **remove-duplicates**. Another is *adjective-noun*: **simple-error**, **double-float**, **direct-subclasses**. A third is *object-attribute*: **object-class**, **slot-allocation**. Note that one should not use the pattern *object-verb*: do not say **file-delete**.
- We need a consistent policy on when to use *attribute* (e.g. **size**) and when to use *object-attribute* (e.g. **file-size**).

The Dylan environment could enforce these guidelines (or just make suggestions) by providing tools that:

- Do spelling correction when an unknown word is introduced, and give the user a chance to think about the "right" word (especially for symbols that will be exported).
- Complain when the user chooses a word that is deprecated because a synonym has been chosen as the standard (e.g., if the user types "create" when "make" is the standard).

- Do syntax checking of compound symbols: complain when the user types “file-delete.”

3. The Word Browser

Once we agree to treat words (and not just symbols) as first-class objects, we can start thinking about what operations can be performed on words. We should be able to define new words, define relationships that hold between them, and search for words (and symbols containing words) that meet certain criteria.

It may be useful for the user to have direct access to a hierarchy of words, to browse it for word definitions and relations to other words, and to extend it when it is time to add new words for a particular project. Here is an idea of what a part of the hierarchy looks like:

function

constructive-function

define

export

make

destructive-function

remove

replace

informational-function

check

compare

modifying-function

append!

object

character

number

integer

real

Attached to each word there should be a description (an English phrase describing the word), a set of relations to other words (e.g., subclass, superclass, synonym, opposite) and a set of equivalents or translations to other languages. For example, the opposite of **forward** is **backward**. The translations of **define** into Lisp would include **defun** and **defvar**. The translations of **append** into C would include **strcat**. The translation of **make** into English would include **create**, **form** and **new**.

It takes some effort to define a taxonomy of definitions for all the words in Dylan, particularly if we want to do a good job of translations to English, C, C++, Basic, Common Lisp, etc. I do think the effort will prove to be worthwhile. For the computer languages, the effort is bounded, and for English, there are public domain resources to help (such as WordNet, a 70,000 word semantic network dictionary of English).

Note that there are two metaphors for finding. The more traditional one is the "database query" metaphor, in which we compose queries (based on words and/or substrings) and find everything that matches. Fry introduces the "translation" metaphor in his "Find Symbol" dialog in Emacs Menus. In this metaphor we state something in one language, and find translations for it into other languages. This supports the database metaphor: one can translate from words and/or substrings to Dylan. But it supports other approaches as well: How would this Basic statement look like in Dylan? What would this Dylan expression look like in C? The two metaphors can be put together by thinking of finding as a database retrieval phase followed by a translation phase.

4. The Phrase Browser

So far, we have dealt with the way symbols are broken down into words. But we also want to be able to go up a level, to deal with the way symbols are combined together to form expressions in Dylan, and the way words are combined to form sentences in English. For example, we would like to know that the Common Lisp expression `(float x)` can be translated to Dylan as `as(<float>,x)`, even though we wouldn't want to translate the word **float** by itself to **as**. Similarly, the English phrase "Set of objects that can be updated" translates into Dylan as the symbol `<mutable-collection>`.

The phrase browser is an interface to a database of information beyond the word level. It includes at least the following:

- Templates for all the Dylan statement/expression types.
- Well-documented examples of all the Dylan statement/expression types, either taken from the manual, or from the libraries. There should be a way of saying "show me an example of a nested loop with a non-local exit." Examples include a call and the return value(s), and are part of the derived database, regardless of whether they originate in code, documentation, or separate regression test sets.
- An interface to the derived information database.
- English phrases from the reference manual. The idea is that users may want to search for "constant time access function" rather than just "access." They should be able to do this by starting at "function" (or "access") in the phrase browser (or the word browser, or a top-level menu), and then browsing the children to see "access function," and then browsing the

children of that to find “constant time access function.” The phrases that are worth keeping in the database can be determined either by some complex natural-language system, or more likely by human-generated annotations to the manual.

- English phrases from comments. English text in comments of user-written code and libraries is just as important as text in the reference manual. Unfortunately, it is not feasible to expect users to annotate their comments,[1] so the complex natural-language system is the only help for pulling out phrases here. Fortunately, there is a lot of off-the-shelf public domain code to get started in this area, should we decide to pursue it.
- Perhaps allow importation of reference manuals/libraries for Basic, C, Lisp, etc.

5. The Finder Form

The word and phrase browsers help the user find something when it can be specified along a single dimension, for example, a kind of access function. But often the user has many constraints on what is being sought. The finder form is designed for these cases. It provides an interface for users to say what they want along a variety of dimensions. It will include fields for:

- Word or phrase: Word or words describing what we’re looking for
- Scope: Language or Document set to look in.
- Module: Dylan module to look in.
- Documentation: Words anywhere in the documentation.
- Example: An example usage taken from the diagnostic suite (which may have originated in a comment or in documentation).
- Purpose: Documentation (or comments) should be structured with a purpose field.
- Author: Name of the author of the desired object.
- Date created/modified: a date or range (earlier/later than) can be specified.
- Sub/Superclasses: As specified in the derived database.
- Calls/Called by: As specified in the derived database.

6. Context-Sensitive Help

Fry’s Emacs Menus (or 4D menus) are a good example of showing the user just what they need. It is important to ease new users into the system, to avoid showing them a blank screen with no idea what to do next. A menu of common top-level forms can go a long away to alleviate this problem, and can give us the best of syntax-directed editors and text-oriented editors.

---

1. Except for a few well-chosen bits of information: perhaps whether the comment is for a “user” or a “hacker,” and perhaps such structured fields as “purpose.”

7. Task-Sensitive Help (The Task Menu/Browser)

The problem with most help systems(even context-sensitive help) is that they document objects on the screen, but don't do a very good job of documenting **actions** or **processes**. That is, help can tell you about a Dylan symbol, or about a button on a tool, but it doesn't help you edit-compile-test-and-debug your program. Task-sensitive help is designed to (a) give you a menu of commonly performed tasks, (b) show you the steps of the task (i.e., the scenario), in particular what you have to do next, and (c) keep track of what you have done and what remains to be done.

The user interface issues need to be sorted out, but I imagine that DylanWorks comes up with a small window (or decoration on an existing window such as the editor) that says "Write Program," and has below it the steps "Edit," "Load," "Test" and "Debug" or something like that. Each of the steps has submenus for substeps, and when you click on, say, "Edit," it brings up the editor and displays a message saying "Write some code-you might want to consider ..." Perhaps an arrow points from the "Edit" subtask to the editor window, a la Apple Guide.Perhaps there is a "Show Example" option, whcich when clicked on runs an "animation" which goes through an example program, step by step, showing the relationships between the tasks on the menu and the various tools that are used, simulating key strokes and mouse clicks.There is also a menu of other tasks that can replace (or be added to) the "Write Program" task. Possibilities include "Send Bug Report," "Patch System," "Write test case," "Write Documentation," etc. There might also be specializations such as "Write Real-Time Interactive Graphics Program." It would be nice if we could easily create tutorials from suitably marked-up source code.

Automatic consistency tests report any errors or warnings when the user tries to do a step that is not ready. For example, if you try to run a program, it may say "The following methods/variables are not defined yet. Shall I define default implementations?" As mentioned elsewhere, there should be a way to record a sequence of actions and encapsulate them into a task menu.

By making tasks first-class objects, we can do some aspects of project management. The manager sends an email message to a programmer that includes a representation of a task object to "code the froggle system and integrate it with bloggles." The programmer loads the object into his/her environment, and begins to work on it, and mails it back when done (or perhaps everything is kept in one big database and there is no need for mail). Similarly, bug reports are first-class objects, and one can ask for all the unresolved bug reports, ask if bugs that were fixed in previous releases are still fixed in the newest release, ask if somebody else is already working on a particular bug report, ask what bug reports have nobody working on them or have been unresolved for over a month, etc.