Design-Oriented Programming:
Macro-Driven Literate Programming in Self-Validating PDL

Joel Vaughn
tragicomedy73@yahoo.com

Paul Graham makes a provoking argument for why many software engineers find that Lisp makes them productive and helps them robust, elegant software through “bottom-up design” [9]. He argues that the expressivity and clarity of programming in Lisp are due to its extensibility: “As you're writing a program, you may think ' I wish Lisp had such-and-such operator.' So you go and write it. ... Language and program evolve together. ... In the end your program will look as though the language had been designed for it” [9]. For much of my time as a software engineer for real-time firmware applications, I’ve wanted to apply the same or similar aspects to my area of expertise. For various reasons, I don’t expect these applications to be written in Lisp any time soon.

In some firmware development processes, there can be a telling tension between program design language and source code. (For most of the issues addressed here the distinctions some make between PDL and pseudo-code won’t matter.) Many standards assume that for the PDL to be useful at all, it must be unambiguous. In other words, you are going to have to write the program twice. Once in a language that is somewhere between Pascal and your native language, and once again in a low-level language (probably C or C++). If you are using the waterfall model, you might even end up rewriting your pseudo-Pascal program (which only worked in theory) based on the source code implementation that mas finally made to work (since it made the design testable). Some programmers joke in this case that it would be nice to write a translator that automatically converts working C into correct PDL. Not such a bad idea, but I think I have a better one: Have a translator convert your design language to C, ***but*** allow that language to document your design, rather than merely define the procedures and the order they are called in, so that the translator can check for semantic issues that only the designer knows to look for. I’ll also argue that macros—**which make use of a semantic preprocessor rather than merely a textual preprocessor—**can be a powerful extension of Donald Knuth's idea for Literate Programming [10].

Let me first clarify that I am talking about *user-defined semantic issues*, not semantic issues for the low-level source code. I’ve seen “mixed mode” operations waste an analyst’s time on documenting the “problem” or get replaced with inefficient code (that won’t make the compiler issue warnings), while a real semantic problem slips by unnoticed at compile-time, such as a values assigned to a variable that hasn’t been converted to the right units or resolution. According to the C compiler, there is no type mismatch! And yet, the engineer’s knowledge that the values are incommensurable could in theory be expressed in PDL, and this problem could be found by “compiling” the PDL. Simonyi's Apps Hungarian notation is a method to enable a "human compiler" to catch design errors, by embedding the intentional information into the identifiers (what Spolsky refers to as the "kind" of data as opposed to "type") [14]. This meta-data could be a part of the PDL itself and these sanity checks could be automated. Some examples of this particular application in CQUAL [7] and other proposals for qualified types (e.g., [3])

When I use a macro, I am often dealing with a problem that can’t be reduced to a function/procedure, either because of real-time efficiency considerations (e.g., stack space, processing time, etc.) or because it is not simply a set of instructions that I am want to abstract. The key point, whether I am using #define to define a constant or conditional compilation directive (for #ifdef) or a field selector or a function-like macro or a construct that alters control-flow, I am attempting to abstract some part of the design. I’m introducing meta-data into the source code. George Polya stated, “An important step in solving a problem is to choose the notation. … The time we spend now on choosing the notation may be well repaid by the time we save later by avoiding hesitation and confusion. … [It] may contribute essentially to understanding the problem” [11] I create notation one #define at a time. Charles Simonyi (in reference to his idea for manipulating design information using a “language workbench”) notes, “In languages such as C, much of the 'intentional' information [i.e., design information] is encoded in macros” [10].

I think this could naturally be implemented as an extension of literate programming. Knuth proposed a scheme for embedding source code in the context of explanatory text (with the use of a markup language--TeX being Knuth's choice) [10]. Source code files are populated with the embedded source code through a "tangling" application (while various design documents may be produced by a "weaving" application). By why limit the design to being defined in terms of the source code? The engineer could define his own notation, and the act of "tangling" can generate source code by expanding these macros, checking their semantic “intentional” information for consistency in the process.

This translator, for which the target is the source code and the source is PDL, will have access to all kinds of meta-data the programmer puts into PDL and each macro expansion will be subject to consistency checks that are part of its definition. You could write macros that can only be expanded (i.e. resolved to a source code tokens or toother macro invocations) inside a for loop, for instance. You could define a new construct and restrict pointer arithmetic to the scope of that construct (resolving the PDL to code would then flag a "tangle-time" error if pointer arithemetic occurred outside one of these user-defined constructs).

But this just scratching the surface. Many firmware platforms do not have the luxury of a full operating system. The limited needs of the application make the interrupt-driven interactions between various communications, signal processing, fault monitoring, and various other background tasks a tractable but tricky problem. From experience, resource management often takes the form of an Executing-Around pattern. This is conceptually related to critical region construct, specifically, and to the RAII design pattern, generally. Raymond Chen's warnings about macros essentially re-defining the language [2] should be taken seriously, but as a warning to not arbitrarily change the alter program flow in a way where it isn't clear what the intended behavior is. A macro that embodies semantic information about its use, that encapsulates intentional information about the software architecture, becomes a notation to help the programmer think about the system. In his work leading to the language constructs of conditional critical regions and monitors, Per Brinch Hansen was looking for "a notation which explicitly restricts operations on data and enables a compiler to check that these restrictions are obeyed" [1]. A PDL can capture these restrictions, and the translator can check them at tangle-time. The notation becomes part of a Domain Specific Language for expressing the design intentions.

I’ve noticed that conditional compilation statements are one way that programmers show how design concerns affect implementation. (Simonyi points this out as well in [12] in his discussion on #ifdef’s.) This is often frowned upon in safety-critical standards, the idea being that engineers could be confused over which statements are actually going to get executed. But bad as counterfactuals might be for implementation, counterfactuals are the essence of design. The particulars of implementation might need to change based on several contingencies for which the effects of a design change can ripple across module boundaries (in spite of encapsulation and attempts to decouple parts of a complex system). The *why* of design is based on contingent decisions. Why must the contingent versions of a module exist only in the designer’s mind, rather than in the documented design? It sounds as though someone is betting on the same developer being around forever (and having a flawless memory). Practical programmers know better. This is why the “static if” and “version” constructs fit well with the pragmatic sensibilities of the D programming language [4].

Macro definitions would, as for C preprocessor macros and C++ template definitions, include invocations of other macros, which would eventually resolve to tokens with semantic information (as

in Dmitriev's templates [6] or template mixins in the D language [5]) and meta-statements and assertions that provide semantic information to nearby statements (what Simonyi calls annotations and stipulations [12]). These macro invocations are more than just abstractions of executable statements. They can contain assertions of how properties and states are changing in the system. Property M of the program has value A after macro X has been invoked (whether macro X eventually resolves to line code or a subroutine call), and macro Y might need to be implemented a different way in source code when M = A, if only to remove redundant "dead code" from the source code. These kinds of implicit dependencies exist in code, but they can be missed in spite of elaborate comments (or because of them).

Simonyi also discusses this possibility that an intention (in this case, an intended behavior) might be be mapped to more than one possible implementation depending on assertions (in his terminology, annotation-dependent application of "reducing enzymes" or intention-to-code mappings) [12]. One ramification of this is that there is a means to address **cross-cutting concerns (or "aspects")** that are represented as annotations in the PDL. The rules for which reduction rule to apply to the macro expansion could be considered aspect "advice," with the macro invocation setting up a well-defined pointcut. Simonyi et al. allude to this use when they claim, "Language extension should be considered … when a particular concern (or aspect) is fragmented over many parts of the source" [13].

Having the PDL translator (while "tangling" or "reducing") check the consistency of meta-statements and assertions in the PDL would combine the power of design validation—as might be expressed in compile-time assertions—with a means of providing aspect-oriented *design*. The engineer builds these self-checks for design assumptions and concerns into the design. Once all the macro substitutions are performed the result is a string of symbols that would , based on the chosen target (say, C source code for a specific DSP), be mapped to specific output text in the target language (C, in this case).

More importantly, design constraints and design assumptions could be first-class entities in PDL, could declared in a PDL-level procedure, and could be identified as being consequences of other decisions and assumptions. Rather than simply grep the source code when I want to check some properties of the design, I could go a step further and be able to query the design itself about why a certain constraint is imposed on a function/module/procedure. **Ideally, a design would document the hard-won knowledge of the engineer about why a system is implemented in a certain way. But that sounds an awful lot like having an expert system embedded in your design document.** So it does. And if another constraint were imposed (or relaxed) by the customer or the system engineers, the expert system embodied in the design document could actually assist the developer in thinking through the ramifications of that change.

Simonyi's ideas for generative programming based on his workbench approach to design intentions suggests a further implication. If there is a design one wishes to apply to very similar applications (say, programs for controllers that solve the same basic problem but with different system parameters and design constraints), you could have one set of design files that produce the various applications with changes to a small set of parameters. **Now the PDL in your design document defines a fourth generation programming language.**

It should be noted here that in many ways, most of these ideas overlap significantly with Simonyi's ideas for intentional programming and Ward's ideas for language-oriented programming [15, 6]. Theidea proposed here of combining literate programming with the use of DSLs to represent design concerns doesn't rule out the "language workbench" approach discussed by Simonyi and also by Fowler [8], any more than the use of markup language precludes a high-level interface for that markup. I'm advocating that a more basic step is that implementing intentional programming using symbol processing as an extension of the literate programming concept, and that macro expansion is a natural way to implement it.

The reason I think that a text-based approach is more fundamental is that Knuth's ideas keeps a very important fact in mind: The act of programming is in a fundamental sense an act of writing. This is why I think Knuth's proposal for literate programming should not be trivialized as merely a neat alternative to commenting source code. In general, figures and charts supplement text, rather than the other way around. A natural medium for a PDL is hypertext (not necessarily bound to a particular markup such as HTML or TeX), because formatting and links are also intentional information; in design commentary, presentation ***is*** content. Any higher-level approach to manipulating design information should not treat commentary text as second-class data. Even charts and tables tend to be convenient representations of text. I think the primacy of text is one reason why there are so many visual plug-ins available for Microsoft Word; it's often more convenient to shoehorn visual presentations into a text processor than to make a presentation-specific application (e.g. a spreadsheet app) deal with arbitrary amounts of text in a smooth comfortable way.

To summarize, I am suggesting that a text-based PDL could be an extremely powerful design tool by allowing a ***notation*** that flexibly handles the following as first-class entities: (a) marked-up expository text (i.e. be literate), (b) user-defined expansion of paramerized symbols into text (be bottom-up and abstract), and (c) embedded meta-data that can be used for code generation and design-validation (be intentional). It could change the way we think about design.

[1] P. Brinch Hansen. Operating Systems Principles. Prentice Hall, 1973.

[2] R. Chen. A rant against flow control macros. `http://blogs.msdn.com/oldnewthing/archive/2005/01/06/347666.aspx`.

[3] B. Chin, S. Markstrum, and T. Millstein. Semantic Type Qualifiers. June 005. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation.

[4] Digital Mars. Conditional Compilation. `http://www.digitalmars.com/d/2.0/version.html.`

[5] Digital Mars. Template Mixins. `http://www.digitalmars.com/d/2.0/template-mixin.html`.

[6] S. Dmitriev. Language Oriented Programming: The Next Programming Paradigm. November 2004. `http://www.onboard.jetbrains.com`.

[7] J. Foster, et al. CQUAL User's Guide. Version 0.991. April 2007.

[8] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? `http:// martinfowler.com/articles/languageWorkbench.html`.

[9] P. Graham. Programming Bottom-Up. `http://www.paulgraham.com/progbot.html`.

[10] D. E. Knuth. Literate programming. The Computer Journal, 27, 97-111, May 1984.

[11] G. Polya. How to Solve It, 2nd Ed..

[12] C. Simonyi. The Death of Computer Languages, The Birth of Intentional Programming. September 1995. Microsfot Research Technical Report MSR-TR-95-52.

[13] C. Simonyi, M. Christerson, and S. Clifford. Intentional Software. In OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications.

[14] J. Spolsky. Making Wrong Code Look Wrong. `http://www.joelonsoftware.com/articles/Wrong.html`.

[15] M. Ward. Language Oriented Programming. Software - Concepts and Tools, 15, 147-161 1994, `http://www.dur.ac.uk/martin.ward/martin/papers/middle-out-t.pdf.`