Markup Language Processing Languages — Where They've Gone Right, And Where They've Gone Wrong

Sam Wilmott

October 5, 2004

Abstract

Basic concepts that seem difficult for the average programmer to comprehend can be packaged in a domain-specific language making their use easy for neophyte programmers. A good example of this kind of packaging of coroutines is the two text processing languages, Hugo and OmniMark. As well as illustrating a model that still seems to be of use in the markup language and text processing field, these languages illustrate a number of things that can go right and wrong when developing domain-specific languages: most importantly, the limits of domain-specific languages.

1 Background

Over the last 30 years I've been involved with designing and implementing text processing programming languages in industry. In the late 1970's I designed and lead the implementation of a programming language, called Hugo, at Canada's Printing Office (the Queen's Printer). Hugo was designed to take input as described by the programmer, and typesetting it as requested. The language continued to be used through the 1980's and finally came to the end of its life when publishing moved away from mainframes to desktops.[1]

[1] There are a variety of programming languages called "Hugo". None of the references I could find using a Google search referred to this language. That's what I get for having predating the Web.

In the late 1980's and early 1990's I designed and participated in the implementation of the OmniMark programming language[2]. OmniMark was designed to take both user-described and standard markup (initially SGML, later XML) input and produce user-described or marked up output. It is used in a wide variety of applications, from batch publishing to on-line document presentation.

2 Hugo and OmniMark

Both these languages share a great deal in common. (Not surprisingly, I can almost hear you say.)

Both languages focus on the problems inherent in in processing and converting textual "documents", originally for print applications, and more recently in conjunction with data base, interactive and web-based services.

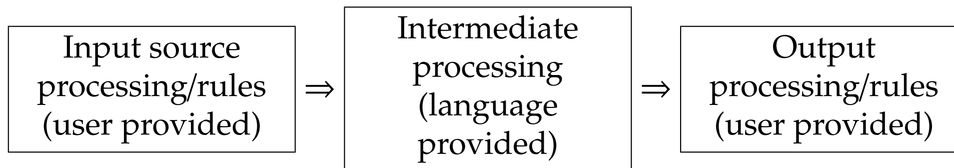
Both languages have "rules" as their basic feature. Because one of the primary functions of both languages is to read and process data of arbitrary kinds, both have rules that read text from an input source and apply patterns to them. The pattern sublanguage in both is of the SNOBOL/Icon language family, rather than the grep/Perl style.

As an example of the pattern matching style, here's a bit of TeX file processing using an OmniMark input rule:

```
find "\begin{" any** => start-style "}"
   start-group (start-style)
   catch end-group (end-style)
      end-group (start-style)
      report-error "mismatched styles: \begin{%g(start-style)}"
                || " vs. \end{%g(end-style)}"
         when end-style != start-style
find "\end{" any** => end-style "}"
  throw end-group (end-style)
```

Both languages also have other kinds of rules corresponding to the primary structures found in their application domains: for Hugo, the other rules correspond to pages, and for OmniMark, the other rules correspond to the elements and other structures found in SGML and XML. Both languages have a built-in program structure, consisting of three synchronous processes, each feeding data into the next one sequentially. A diagram that describes both is the following:

[2]see <http://developers.omnimark.com>.



For the Hugo typesetting language, intermediate processing is the "hyphenation and justification" phase of typesetting (commands specifying paragraph boundaries and typesetting characteristics are issued by the user in the input phase), and the rules provided for output processing correspond to page layouts. For the OmniMark language, intermediate processing is (typically) SGML or XML markup language processing,[3] and the rules provided for output processing correspond to the structural components of the marked-up input: elements etc. In OmniMark, rule invocation can be nested: elements occur within elements. Note that rules are not performed for structural boundaries, as in some other SGML/XML markup processing techniques, rather a rule corresponds to a structural item, such as an element, as a whole.

Running the three phases of processing in parallel — as mutually synchronous coroutines rather than one at a time — means that feedback from later phases can be used in earlier phases. For example:

- In a Hugo program, one can determine what page typesetting has got to while processing further input. This information helps in certain typesetting situations.
- In an OmniMark program, one can determine where one is in the structure of an XML document produced by converting from another form. This information provides context for the conversion.

In both cases, there's a certain asynchronicity inherent in the process, so feedback is not always accurate. However, there are various ways, and certain conditions in which feedback is accurate and can be made use of. For example, local synchronization occurs when it's known that a full markup token has been fed to the the XML parser, or that a "flush" has been done to the current typeset page.

[3]OmniMark supports intermediate processing for other syntactic forms, such as RTF, but as "plug-ins" implemented in C or C++.

Providing language features that exploit coroutines rather than forcing the user to figure out what to do with them means that relatively inexperienced programmers can benefit from them. Understanding the roles of different programming language features is important when designing programming languages:

- There are features such as coroutines, whose primary role is in the implementation of other, higher-level features. Coroutines may not be of direct use to the average programmer, but they can greatly ease implementing tools and languages used by the average programmer.
- There are features such as rules, whose primary role is to provide users with a convenient interface to domain specific functionality. What's needed on the tool side if convenient functionality for implementing specific kinds of rules.

3 Strengths And Weakness

Domain-specific languages typically address the needs of a specific audience. This is their strength, but also their weakness. This leads to initial and ongoing success within that audience: they see how the language addresses their needs and improves their productivity, and they become familiar with the language

Where the audience is well defined, and where its needs stay largely stable, as in the case of the Hugo programming language, the implementation and support of the language are well worth the cost. On-going support typically requires a small staff, which can be usually justified within a medium to large organization, where the language's benefits are significant, even if small.

On the other hand, there are many things that can cause a domain-specific language to fail:

- The audience is not well defined, or is lost sight of.
- The implementors are not in direct contact with the needs of their audience.
- The needs of the audience change beyond what can be readily done or added to the language.
- In an attempt to increase functionality, features are added to the core language rather than in libraries — resulting in incompatibilities between languae versions.

- Other technologies emerge that are perceived to fill the same need set, and which are more appealing for cost, "popularity" or functionality reasons.
- There isn't a direct benefit from the language for the organization supporting it (i.e. where it isn't substantially used within that organization).
- The initial success of a domain-specific language misguides those in control of it, so that they think they have in hand more than they really have.
- The language implementation, originally designed with one domain in mind, doesn't lend itself to adaptation to different goals. This is, of course, a difficulty with any piece of software — it does what it is designed to do, and even if good at that, doesn't always lend itself to ready support of other functionality.
- In general, it is far too easy for a domain-specific programming language to evolve into a general purpose programming language — general purpose and common programming features are the easiest to understand and the easiest to see how to implement. The result is all too often a good domain-specific programming evolving into a bad general purpose programming language.

All of these factors, together with the truth that all but the most basic of technologies have a finite lifetime, have contributed to OmniMark's current reduced popularity, as compared to ten years ago.

A key factor in the success of and programing language in support. For a popular general-purpose programming language, this support can be found in readily-available books and in the educational system. This support is not generally available for a domain-specific language, and has to be provided in other ways, typically by the the organization . As well there are support requirements unique to domain-specific programming languages: domain-oriented training, aid in developing applications, and incident help for when things go wrong.

Domain-specific programming languages can be a strange experience from the point of view of the typical computer programmer. Such languages can be more accessible to non-programmers, who have not developed language-specific expectations. In fact, designing for non-programmers ("human beings" I've often called them in contrast with experienced programmers) can produce a programming language that is unfamiliar to

programmers experienced in main line programming languages. Both audiences need a proactive support service for the domain-specific language:

- the human beings because they are new to using programming languages, or at least have limited experience, and
- the others, whose experiences lead them astray in their expectations and in their use of the language.

What's not often well understood is the level of support required by otherwise experienced programmers. Just recently I was talking to a programmer who had attempted to use OmniMark and "just couldn't figure it out". He was of the (no doubt correct) opinion that had he be able to attend a course on the language, his experience would have been different. On the other hand I've talked to many users of Hugo or OmniMark who had no previous programming experience but who, with a minimum of introduction, find their use "natural". They say things like "you try something and it does what you expect it to do", and "it's easy when you can process data 'on the fly' ".

4 Developing Domain-Specific Languages In Industry

Domain-specific programming languages are developed both in the academic community and in industry. Each have their problem domains where domain-specific programming languages can help.

Industry has the advantage that the language designers and implementers are often closer to problems where non-expert programmers (although typically expert in the domain) can benefit from domain-specific programming languages. For these users, domain-specific languages may be the first they ever use. Industry has the added advantage that there are often the resources available to devlope and support a domain-specific programming language.

The academic community has the advantage that it's more likely that those using the domain-specific programming language will retain control over its development. The down-side to industry, is that technical decisions tend to get made at a management level, to the detrement of all.

5 The Role Of Coroutines

Coroutines are a key part of both Hugo and OmniMark, and I believe in a wide range of other kinds of text processing processing. In fact, I'd go so

far as to say:

- Markup language processing and other text processing is currently the most important and widespread class of programming applications.
- General markup language and text processing applications typically have to perform a variety of tasks, many of which can most conveniently feed their output to other such tasks. i.e. They are coroutines, although may not be perceived as such by their users.
- Any general programming language used for the implementation of markup language and text processing functionality, that doesn't have good support for coroutines, isn't good enough.

If there's one tool that's made implementing domain-specific programming languages and their components — like XML parsers — difficult, it's the absence of coroutines in the programming languages used in their implementation.

Why aren't coroutines more widely supported? Here's my take:

- People don't have experience with them.
- Current explanations of coroutines are all too often attempt to model them on continuations. Whenever "continuations" are mentioned, most programmers, not unreasonably, run the other way.
- Coroutines are a fundamental, general language feature. It's often difficult to retrofit them to an existing programming language implementation, especially when performance or reliability are concerns.
- Coroutines are very useful for implementing tools for use by other programmers, like markup language parsers. They are generally of less use for casual programming tasks. As a consequence it's often hard to explain why they're needed: the counterargument to needing them is often "I don't need them."
- Existing "main stream" programming languages tend to be "data oriented": they have lots of facilities for defining and using data structures. Control structures have tended to take a second place. One could even say "Object-oriented programming considered harmful" — at least as they've focussed language development exclusively on data structures.

- Both Windows and Linux make use use of the machine's stack pointer (the SP/ESP register on Intel machines) for purposes other than program flow. Linux uses the stack pointer for locating "thread global data". Windows, it seems, uses the stack pointer in conjunction with allocating heap memory. Such uses make efficient implementation of coroutines difficult.

6 The Other Way

The most popular programming languages for markup and text processing are currently:

- General use languages like C++, Java and C#, with special use libraries for markup and text processing.
- Text processing languages like Perl and Python, with with a mix of language-provided features, like Perl's pattern matching, and special use libraries, especially for markup language processing.
- Markup specific languages of the sort provided by the W3C[4]. These languages have the markup language processing "built-in".

All these languages and their processing models emphasize static data processing. Markup language data, for example, is typically mapped into an annotated tree structure, and tools are provided for walking the tree. In contrast, Hugo and OmniMark map use this kind of structure drive the flow of processing within programs – in effect, mapping structure into dynamic control structure rather than static data structure. The difference, in practice, is that the programmer is faced with much less detail in the data they need to work with. This makes these languages much more accessible to neophyte programmers for this kind of processing, than are many of the other approaches.

7 Domain-Specific Sublanguages

One conclusion I've come to out of my experience is that as far as possible, domain-specific functionality should be implemented as sublanguages of

[4]see <www.w3.org>

existing, more general programming languages. This isn't always possible of course. And neither existing programming languages nor their implementations help.

I think a good way to identify what's wrong with existing programming languages vis-à-vis implementing domain-specific languages is to twist the question around and ask why we can't add that functionality to an existing language. Part of the answer comes out as follows:

- Coroutines. OK, I've already said that.
- Defining mechanisms for application-specific rules.
- Good overloaded operator facilities. Operators are not just for numbers.
- Strong dynamic language facilities.
- A strong and general static type model. Dynamic languages are great for casual use, they provide useful tools for implementing language features in the languages themselves, and they make a great basis for good domain-specific languages. But you really also need static typing to efficiently implement tools for programmers. These are not incompatible requirements.

This isn't the whole answer though:

- Such a dream language, with all these capabilities, doesn't exist.
- There's the problem of syntax. You sometimes just need to use some very domain-specific terminology and forms of expression and it's never going to fit into even the dream language. Actually though, it makes sense to design a family of languages, all with the similar underlying functionality, but with appropriate syntax for different classes of applications.
- Domain-specific languages typically have major functionality "behind the scenes". It's not clear that all such functionality can be incorporated in an otgherwise "normal" programming language.

So what next? There needs to be more literature on implementing domain-specific languages. And in a form accessible to those in industry who are implementing domain-specific languages in daily use. But lot more work too – application-friendly programming languages are not a solved problem.