CSci 658: Software Language Engineering
Domain Specific Languages

H. Conrad Cunningham

2 April 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall

P.O. Box 1848
University, MS 38677
(662) 915-5358

Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of April 2018 is a recent version of Firefox from Mozilla.

TODO:

- Finish missing sections
- Structure the document better
- Integrate example external and internal DSLs with this document with whatever language(s) are needed
- Better integrate the various lists of guidelines for construction of DSLs. Better tie these to concrete examples
- Consider monadic DSLs in Haskell, macro systems, Template Haskell, staged metaprogramming, etc.

Domain Specific Languages (DSLs)

What are DSLs?

Motivation

Few computer science graduates will design and implement a general-purpose programming language during their careers. However, many graduates will design and implement—and all likely will use—special-purpose languages in their work. These special-purpose languages are often called domain-specific languages [Wikipedia].

Paul Hudak describes a *domain-specific language* (or *DSL*) as "a programming language tailored to a particular application domain" [Hudak 1998], that is, to a particular kind of problem.

General-purpose languages (GPLs), such as Java, Python, C, and Haskell, seek to be broadly applicable across many domains. They can, in theory, compute any function that is computable by a finite procedure; they are said to be *Turing complete*.

DSLs might be Turing complete, but often they are not. DSLs are *little languages* [Bentley 1986] that "trade generality for expressiveness" [Mernik 2005].

Ideally, a DSL should enable experts in an application area to program without programming—that is, to express the problems they want the computer to solve using familiar concepts and notations, without having to master the intricacies of programming in a general-purpose language [Hudak 1998, van Deursen 2000].

Examples

As discussed in Bentley's classic column on "Little Languages" [Bentley 1986], the DSL `pic` enables writers to produce line drawings in typeset documents; they can focus on the layout of the drawings without being required to develop programs in C, the primary general-purpose language used on Unix. (See Bentley's column on little languages [Bentley 1986] for discussion of this DSL.)

The `pic` language and tool were built according to the Unix philosophy. This approach focuses on a minimalist and modular approach to software development. The Unix tools have limited functionality but are usually built to read and write streams of bytes so that tools can be readily composed using Unix pipes.

Other DSLs on the Unix (or Linux) platform include:

- `awk` for extracting data and generating reports from streams of text-based data using regular expressions to specify the processing
- `lex` for generating lexical analyzers
- `yacc` for generating parsers
- `make` for building software from its various sources
- `cpp` for preprocessing C programs to include header files, expand macros, and support conditional compilation
- `dot` little language for specifying graph structures within the Graphviz set of tools

Markup languages are also DSLs.

- HTML is a DSL for formatting documents on the Web.
- LaTeX is a powerful markup language used for books and articles (especially in STEM disciplines).
- Markdown is a simple markup language for documents that may be needed in various formats, especially HTML. It is often used by wikis and other websites with user-contributed content (e.g. Wikipedia, GitHub). This document is written using the variant of Markdown supported by the Pandoc tool.
- reStructuredText is a simple markup language used mostly in the Python community for technical documentation.

The designers of a DSL must select relevant concepts, notations, and processes from the application domain and incorporate them into the DSL design [Hudak 1998].

Definition

Martin Fowler defines a *domain-specific language* as a "computer programming language of limited expressiveness focused on a particular domain" [Fowler 2011].

He explains these terms as follows:

- As a **language**, a DSL has *fluency*. The expressiveness of the language comes not just from the simple expressions but also from how those expressions can be easily composed to form larger units.
- As a **computer programming language**, a DSL is structured so that humans can effectively read and write "programs" in the language and computers can accurately read and "execute" the instructions. It has a well-defined syntax and semantics.
- Having **limited expressiveness** means the DSL includes features that are needed for its purpose and excludes features that are not needed for that purpose.
- Being **focused on a particular domain** means that the DSL has a purpose that is both clearly and narrowly defined.

Boundaries

What is and is not a DSL is a fuzzy concept. Not all writers agree on the definition. In these notes, we use Martin Fowler's definition above to consider whether or not a notation is a DSL.

Fowler suggests that one key characteristic of a DSL is its *language* nature [Fowler 2011].

- The language (DSL) is intended to be read and written by humans as well as read by computers.
- The DSL must be *fluent*. That is, the meaning of the language comes not just from the individual expressions (i.e., the words) but also from how the expressions are composed together (i.e., into sentences and paragraphs). It has both a vocabulary and a grammar.

Fowler also suggests two other key characteristics of a DSL—its *limited expressiveness* and its *domain focus* [Fowler 2011].

He argues that a DSL should have only those features needed to support its target domain. A DSL should not attempt to solve all problems for all users for all time. It should not seek to define an entire software system. It should instead focus on providing an effective, uncluttered solution to a specific aspect of the overall system.

Which of these boundaries is important in a particular situation depends upon the style of the DSL.

External and Internal DSL Syles

Fowler classifies DSLs into two styles [Fowler 2008a, 2011]:

- external
- internal

Although the terminology is relatively new, the ideas are not.

External

An *external DSL* is a language that is different from the main programming language for an application, but that is interpreted by or translated into a program in the main language. The external DSL is a *standalone* language with its own syntax and semantics.

The Unix little languages `pic`, `lex`, `yacc`, and `make` exhibit this style. They are separate textual languages with their own syntax and semantics, but they are processed by C programs (and may also generate C programs).

External DSLs may use ad hoc techniques (e.g. hand-coded recursive descent parsers), parser-generation tools (e.g. `lex` and `yacc` in the C/Unix environment, Happy and Alex on the Haskell platform, ANTLR on various platforms), or parsing libraries (e.g. Haskell library Parsec, Scala parser combinator library, Python 3 parser combinator library Parsita, Lua LPeg library).

Example external DSLs in the course notes include

- most of the Scala-based State Machine (Secret Panel) DSLs (adapted from [Fowler 2011])
- one of the Lua-based Lair Configuration DSLs (adapted from [Fowler 2008b])
- three of the Ruby-based Reader DSLs (adapted from [Fowler 2008a])

To distinguish between external DSLs and GPLs, Fowler cites the need for a DSL to limit its features to the minimal set needed for the specific domain [Fowler 2011]. (See discussion in Boundaries subsection above.)

For example, Fowler considers the programming language R a GPL not a DSL. Although R has special-purpose features to support statistical programming, it has a full set of general purpose features for a wide range of programming tasks.

Internal

An *internal DSL* transforms the main programming language itself into the DSL–the DSL is *embedded* in the main language [Hudak 1998].

The techniques for constructing internal DSLs vary from language to language.

The language Lisp (which was defined in the 1960s) supports *syntactic macros*, a convenient mechanism for extending the language by adding application-specific features that are expanded at compile time. The Lisp macro approach has been refined and included in languages such as Scheme, Clojure, and Elixir.

Internal DSLs in the language Ruby exploit the language's *flexible syntax*, runtime *reflexive metaprogramming* facilities, and *blocks* (closures). The Ruby on Rails web framework includes several such internal DSLs.

Haskell's algebraic data type system has stimulated research on "embedded" DSLs for several domains including reactive animation and music [Hudak 1998, 2000].

In object-oriented languages, internal DSLs may also exploit object structures and subtyping.

Example internal DSLs in the instructor's notes include the following:

- Scala-based Computer Configuration DSL (adapted from [Fowler 2011])
- Scala-based Email Message Building DSL (adapted from [Fowler 2011])
- Most of the (Lua- and Python-based) Lair Configuration DSLs (adapted from [Fowler 2008b])
- Ruby-based Survey DSL [Cunningham 2008] (partially motivated by the sidebar in [Bentley 1986])
- Haskell- and Scala-based Sandwich DSL

To distinguish between an internal DSL and a standard *command-query Application Programmer Interface (API)*, Fowler cites the need for a DSL to be fluent [Fowler 2011]. The command-query API provides the vocabulary; the DSL provides the grammar for composing the vocabulary "words" into "sentences".

The implementation of a DSL is often supported by an API with a *fluent interface*, a vocabulary of operations designed to be composed smoothly into larger operations.

Shallow and Deep Embeddings of Internal DSLs

The difference between shallow and deep embeddings of an internal DSL concerns the relationship between the implementations of a DSL's syntax and its semantics.

Shallow embedding

In a *shallow embedding* of an internal DSL, the implementation's types and data structures directly representa single interpretation of the semantics of the domain but do not represent the syntactic structure of the domain objects.

For example, the regular expression package from the Thompson Haskell textbook [Thompson 2011], section 12.3, is a shallow embedding of the regular expression concept. It models the semantics but not the syntax of regular expressions. It uses functions to represent the regular expressions and higher order functions (combinators) to combine the regular expressions in valid ways.

Similarly, the Scala-based Computer Configuration and Email Message Building DSLs and the Lua-based Lair DSLs are relatively shallow embeddings of the DSLs.

The advantage of a shallow embedding is that it provides a simple implementation of the semantics of the domain. It is usually straightforward to modify the semantics by adding new operations. If these capabilities are all that one needs, then a shallow embedding is convenient.

A disadvantage is that it is sometimes difficult to relate what happens during execution to the syntactic structure of the program, especially when errors occur.

Deep embedding

In a *deep embedding* of an internal DSL, the implementation's types and data structures model both the syntax and semantics of the domain. That is, it represents the domain objects using *abstract syntax trees (ASTs)*. These can be interpreted in multiple ways as needed.

For example, section 19.4 of the Thompson Haskell textbook [Thompson 2011] redesigns the regular expression package as a deep embedding. It introduces types that represent the syntactic structure of the regular expressions as well as their semantics.

The advantage of a deep embedding is that, in addition to manipulating the semantics of the domain, one can also manipulate the syntactic representation of the domain objects. The syntactic representation can be analyzed, transformed, and translated in a many ways that are not possible with a shallow embedding.

As an example, consider the deep embedding of the regular expression DSL. It can enable replacement of one regular expression by an equivalent simpler one, such as replacing `(a*)*` by `a*`.

Of course, the disadvantages of deep embedding are that they are more complex to develop, understand, and modify than shallow embeddings.

Expression Language

What about the Expression Language (language definition, parsing, compiling) discussed in a separate case study?

The concrete syntax and semantics of the Expression Language is different from its host language Haskell, so at that level it is an external DSL. The parsers recognize valid arithmetic expressions in the input text and create an appropriate abstract syntax tree inside the Haskell program. The abstract syntax tree differs from the textual arithmetic expression and from its parse tree.

However, the abstract syntax tree does capture the essential aspects of the syntax. And the abstract syntax tree itself can be considered a deep embedding of an internal DSL for the abstract syntax. For the remainder of the processing of the expression, the abstract syntax tree preserves the important syntactic (structural) features of the arithmetic expressions.

The Expression Language case study transformed the abstract syntax trees by simplifying them (and by generating their symbolic derivatives). The case study also translated the expressions to instruction sequences for a Stack Virtual Machine.

The accompanying Sandwich DSL case study (in Haskell and Scala) gives another example of how one can create a deeply embedded DSL in a simple situation.

DSLs Used to Produce This Document

When we look around, we can see DSLs everywhere!

Consider how I create this document on DSLs. I write the text with the text editor Emacs, which includes a number of extensions, perhaps written in DSL(s). I indicate the document's structure using a DSL, Pandoc's dialect [MacFarlane 2018] of the markup language Markdown. If I add mathematical notation to the document, I write these in a subset of the LaTeX DSL. I then execute the `pandoc` tool on the input file.

Pandoc (which itself is written in Haskell) reads the Markdown input, converts it to an abstract syntax tree (AST) internally, and then writes an appropriate HTML (a DSL) output file (that you are likely reading). I also direct Pandoc to write a LaTeX (a DSL) output file, on which I execute the tool `pdflatex` to create a PDF document. I could generate documents in other standard formats such as Microsoft Word's `.docx` format (essentially a DSL) and EPUB (a DSL).

For the HTML output, I direct Pandoc to generate MathML, a standard DSL in the XML family that describes mathematical expressions for display on the Web. (It is currently supported by the FireFox browser but not all others, which is why I recommend viewing these documents with FireFox.)

If I wish to include a drawing of a graph structure in the document, I may express the graph in the `dot` language supported by the Graphviz package. Then I can translate it into a Scalable Vector Graphics (svg) drawing, which is encoded with another dialect of XML.

I wrote the original version of this document directly in HTML before I started using Pandoc. Other documents used in my courses were written originally using LaTeX or Word. In some case, I used Pandoc to convert these documents to Markdown, which gave me the starting point for my recent changes.

To add a new input format to Pandoc requires a new *reader* program that can *parse* the input and generate an appropriate abstract syntax tree.

To add a new output format to Pandoc requires a new *writer* program that can access the abstract syntax tree and generate appropriately formatted output.

Pandoc's abstract syntax tree is made available to writers using either Haskell algebraic data types or JSON (JavaScript Object Notation) structures.

JSON is an external DSL in that it uses a subset of JavaScript's concrete syntax to express the structure of the data. But, because it is JavaScript code, it also defines an equivalent internal data structure, so it also has aspects of an internal DSL. Pandoc could use a JSON Schema to define the supported format. A JSON Schema is a JSON document with a specific format (a DSL) that defines the format of other JSON documents (other DSLs).

The whole conversion process in Pandoc revolves around the abstract syntax tree. Pandoc enables users to write their own filters that transform one Pandoc AST to another.

Thus my workflow uses many DSLs directly or indirectly.

Possible Advantages of Using DSLs

Fowler and others give several possible advantages for using DSLs. These include:

1. *DSLs can facilitate communication between domain experts and the programmers.* [Fowler 2011]

   A DSL may be a small, simple language that can express important aspects of the domain in a manner that nonprogrammers can *read* and understand.

   In some cases, users may be able to *write* the DSL programs, allowing them to adapt the application to their specific needs without the intervention of programmers. But designing a DSL that users can write effectively is considerably harder than one they can read effectively.

   This is an attack on the effects of the essential complexity of software development [Brooks 1987].

2. *DSLs can help encode domain knowledge in concise, concrete forms readable by humans.* [Spinellis 2001]

   This seeks to mitigate the problem of deciding what to build, lessening the effects of the essential complexity of software development [Brooks 1987]. It also enables this knowledge to be reused more readily.

3. *DSLs and their implementations can increase opportunities for software reuse.* [Mernik 2005]

   An implementation of a DSL may generate code that incorporate key data types, software architectures, algorithms, and other domain concepts and processes. These are reused from one use of the DSL to another.

   Software reuse is an attack on the effects of the essential complexity of software development [Brooks 1987].

4. *DSLs can improve programmer productivity.* [Fowler 2011]

   Using a DSL for some narrow, well-defined aspect, programmers can code the computation more quickly and reliably with the DSL than directly in the GPL.

   For example, many GPLs have sublanguages for specifying regular expressions. In most cases, programmers can use these regular expression DSLs more productively than programming the pattern recognition directly in the GPL. Also, a mature implementation of the DSL will likely be more reliable than a new implementation directly in the GPL.

5. *DSLs can help shift the execution context between compile time and runtime.* [Fowler 2011]

   On the one hand, if some aspect of an application needs to be more flexible, we might replace a description in the GPL program (e.g. populating a complex data structure) with a description in a DSL that is interpreted at runtime.

   On the other hand, if some aspect of an application needs to be more efficient, we might replace a description of the runtime configuration code with a description in a DSL that is compiled into efficient GPL code.

6. *DSLs can enable convenient use of alternative computational models that are not natively supported by the GPL used in the overall application.* [Fowler 2011]

   For example, some aspect of the computation might better be expressed as a finite state machine, decision table, dependency network, rule-based system, etc. We can embed such computations within traditional imperative or object-oriented programs by designing< appropriate DSLs.

7. *DSLs can promote porting aspects of the application code to a different execution platform.* [Ward 1994]

The front end phases of the DSL need not be changed, just the back end that does code generation or interpretation.

Note: Fowler argues this is a benefit of having a semantic model, not necessarily a DSL.

Possible Disadvantages of Using DSLs

Fowler also identifies several possible disadvantages of using DSLs. These include:

1. *DSLs can contribute to the **language cacophony**.* [Fowler 2011]

   If many, different, difficult-to-learn languages are used in an application or an organization, then this creates considerable work for the software developers to learn them all.

   Fortunately, DSLs are usually much easier to learn than GPLs. Also the abstractions represented in the DSLs likely will need to be present in the libraries, APIs, documentation, and tools regardless. Thus using DSLs might not incur as much language learning as it first appears.

2. *DSLs can be costly to build and maintain.* [Fowler 2011]

   A DSL is usually built on top of a library, API, or software framework. Designing, implementing, and maintaining the DSL will require some additional work.

   Fortunately, DSLs are usually simple and once the developers master the language development tools, designing and implementing a DSL can be done without a huge investment.

   As with any tool development or acquisition, the software development organization must decide if the possible benefits are greater than the costs.

3. *DSL can contributes to the **ghetto language** problem.* [Fowler 2011]

   If a language uses many languages that are used nowhere else, then it can become difficult to recruit staff.

   This can be a significant problem for use of a GPL. (However, some software development organizations choose languages outside the mainstream so they can attract the more aggressive staff willing to take on interesting technical challenges and use leading edge tools.)

   But DSLs should be small and limited to a narrow domain, so the problem should not be as significant as for GPLs. The organization should guard against rampant "mission creep" for its DSLs.

4. *DSLs can sometimes lead to narrow thinking.* [Fowler 2011]

The intention of DSL development is to open up the developers to using whatever abstractions are appropriate to the domain, rather than those that are convenient in the GPL.

Organizations that use DSLs should avoid falling back into the same trap with their DSLs. They should develop appropriate new DSLs when needed rather than use an existing DSL with inappropriate abstractions.

Designing DSLs

In a chapter in the functional programming notes, we examine *families* of related functions to define generic, higher-order functions to capture the computational patterns for each family. We seek to raise the level of abstraction in our programs.

Design of DSLs is similar, except that we seek to design a language to express the family members in some application domain rather than design a higher-order function.

SCV analysis

We should first analyze the domain systematically and then use the results to design an appropriate DSL syntax and semantics. We analyze the domain using *Scope-Commonality-Variability (SCV) analysis* [Coplien 1998] and produce four outputs.

1. *scope* – the boundaries of the domain. That is, identify what we must address and what we can ignore.
2. *terminology* – the definitions of the specialized terms, or concepts, relevant to the domain.
3. *commonalities* – the aspects of the domain that do not change from one application to another within the domain. We sometimes call these the *frozen spots*.
4. *variabilities* – the aspects of the domain that may change from one application to another within the domain. We sometimes call these the *hot spots*.

TODO: Expand the explanation (e.g., include model) to include more of the ideas from [Coplien 1998] and other sources. Perhaps include example such as the SurveyDSL design from [Cunningham 2008].

In the SCV analysis, we must seek to identify all the implicit assumptions in the application domain. These implicit assumptions need to be made explicit in the DSL's design and implementation.

We use the SCV analysis to guide our choices for elements of the DSL design [Mernik 2005, Thibault 1999, Cunningham 2008]. The scope focuses our attention

on what we are trying to accomplish. The terminology and commonalities suggest the DSL statements and constructs. The commonalities also suggest the semantics of the constructs and the nature of the underlying computational model. The variabilities represent syntactic elements to which the DSL programmer can assign values.

DSL design guidelines

TODO: Better integrate the Karsai, Freeman, and other lists of guidelines.

Karsai et al [Karsai 2009] identifies 26 guidelines important for DSL design, grouping them into 5 categories. The paper focuses on design of external DSLs. Here we expand their guidelines to include internal DSLs.

A. Language purpose guidelines (What purposes to satisfy)

   1. *Identify language uses early.*

      In Fowler's terminology, we must identify the domain and what uses the language will have within the domain. We must carefully define the scope within the SCV analysis.

   2. *Ask questions.*

      What group of users will write the DSL programs? will read them? will deploy them for execution? Etc. What are each group's purposes? What does each group need to be able to understand and use the DSL successfully? Can we simply the DSL further?

   3. *Make the language consistent.*

      Avoid surprises. Keep the DSL narrowly focused on its purposes. A DSL feature should contribute to the purposes or be omitted. All features should be based on a cohesive set of concepts.

B. Language realization guidelines (How to implement)

   4. *Decide carefully whether to use a text-based external DSL, a graphical external DSL, or an internal DSL hosted in some particular language.*

      For the DSL's identified domain, uses, and user groups, what are the advantages and disadvantages of each approach? What tools are available to support the DSL design, implementation, and use.

   5. *Compose existing languages where possible.*

      Can parts of the new DSL's uses be handled by already implemented languages and tools? If so, we can avoid the time-consuming and error-prone work of designing and implementing a whole new DSL. We can combine the existing languages, embed them within a new

"glue" language, extend an existing language with a few new features, etc.

In the "Little Languages" paper, Bentley describes how the processor for the external DSL `chem` generates a program in the DSL `pic`. The processor for `pic` itself generates a program in the DSL `troff`. These "filter" programs are then connected using pipes in the Unix shell (a DSL). Furthermore, the implementation of `pic` specifies the lexical analysis and parsing phases using the DSLs `lex` and `yacc` and defines the overall build process using the DSL `make`.

Consider an internal DSL such as the Survey DSL. It includes many existing features of the host language (Ruby) in the new DSL.

6. *Reuse existing language definitions.*

   Even if the implementation of an existing language cannot be reused, we can consider reusing its definition. This saves effort in language design and it may leverage the users' knowledge of the existing language.

   For example, the Pandoc Markdown dialect embeds LaTeX's widely-known mathematical notation to specify mathematical symbols and expressions within the text.

   Sometimes building a DSL entails embedding an existing API within a fluent interface to implement an internal DSL or devising a similar textual notation to form an external DSL [Mernik 2005]. Or perhaps we reenvision a textual or internal DSL as a graphical DSL.

7. *Reuse existing type systems.*

   A language's type system is probably the most difficult to design well and implement robustly. A new type system can also be difficult for users to learn to use effectively. Thus, by reusing an existing type system that the users may know, DSL developers can both make their work more efficient and the new DSL easier to understand and use.

   An internal DSL selects from and builds on the host language's existing type system.

C. Language content guidelines (What features to include)

8. *Reflect only necessary domain concepts.*

   Which artifacts (or objects) from the domain must we capture to satisfy the DSL purposes? Which properties of those artifacts? Can we leave out the other artifacts and purposes? We should discuss possible designs with users and incorporate their feedback.

9. *Keep the DSL simple.*

Make the DSL easy for users to learn and use effectively. If the language is too complex, users may just ignore it. Keep the language as simple as possible to ensure effective use.

Simplicity may be especially difficult to achieve in internal DSLs because the boundary between the DSL features and the host language may not be clear. Seek to crisply define this boundary.

The next three guidelines help us achieve simplicity.

10. *Avoid unnecessary generality.*

    This is an aspect of keeping the DSL simple. Design only what is necessary to solve the problem. Avoid excessive concern about generalizing and parameterizing the language beyond what is needed initially. It is difficult to predict what generalizations will actually prove useful.

    However, a successful DSL is seldom static. It likely must evolve to meet the changes in the domain and the expectations of its users. Design the DSL so that it can be extended with new capabilities in the future.

    An SCV analysis can reveal important "variabilities" (hot spots) that either should be incorporated in the initial design or that may become important in the future.

11. *Limit the number of language elements.*

    This is a second aspect of keeping the DSL simple. A language with many elements is difficult to learn, use, and implement. It is better to have a few elements that can be combined flexibly.

    If the domain and purposes are complex, look for ways to break them into a set of smaller problems, solve each subproblem by designing a sublanguage, and then combine the sublanguages to solve the larger problem. Users can focus on the sublanguages they need to carry out their specific tasks.

    Alternatively, determine whether some of the more complex elements can be moved from the DSL's core and to a "library" that can be accessed by DSL programs. Enable users to store their own language extensions in the library. This approach enables us to extend the functionality of the DSL without changing the core language's structure.

    For example, languages like Pascal included I/O statements as a language construct. Later languages such as C and Java moved I/O to a library.

12. *Avoid conceptual redundancy.*

This is a third aspect of keeping the DSL simple. If there are many possible ways to express the same concept in the DSL, users will likely become confused and may not use the DSL effectively. Avoid unnecessary redundancy.

13. *Avoid inefficient language elements.*

    A DSL raises the level of abstraction and usually obscures the details of how a DSL program is actually executed. However, the DSL designers and implementers must ensure that the DSL programs execute with acceptable efficiency. Moreover, the user of the DSL must be able to understand which DSL programs are more efficient than others.

D. Concrete syntax guidelines (How to make the DSL readable)

14. *Adapt existing notations domain experts use.*

    Where possible, build on the formal notation that the domain experts already know and use. If this needs to be modified or extended, keep the changes close to the style of the existing notation. Familiarity will make the DSL easier for domain experts to learn and

    Of course, DSL designers may need to formalize the syntax and semantics of informal terminology, notation, and processes to enable them to be included in an automated DSL. Tools for processing the new DSL may also provide capabilities not present in the current practice.

15. *Use descriptive notations.*

    Where possible, choose terms and symbols that suggest the intended meaning. Avoid using them in ways that differ significantly from the way they are used in the domain or in the general public.

    For example, the symbol `+` usually denotes addition or some similar operation; to use it to denote multiplication would likely introduce confusion and make the DSL difficult to learn. Similarly, using the keyword `if` to denote something other than a conditional would be confusing.

16. *Make elements distinguishable.*

    A DSL will be read by humans more frequently than written. So, when the needs of readers and writers conflict, favor the readers over the writers.

    Make different elements appear differently in the displayed form. Avoid using a subtle difference in notation, spelling, location, size, font, or color as the sole way to distinguish between different elements. In most cases, make the DSL accessible to those with impaired vision.

17. *Use syntactic sugar appropriately.*

Syntactic sugar refers to elements of a language that do not add to the expressiveness of the DSL but which may make it easier to read and perhaps easier to parse.

If used in moderation, syntactic sugar makes the language more palatable. But overuse may just make the language fat -- more verbose and confusing.

This guideline conflicts with the "avoid conceptual redundancy" guideline above. Designers must balance these concerns.

18. *Permit comments.*

    Although comments do not make the DSL more semantically expressive, comments enable those writing DSL programs to explain their design decisions to others (or their future selves) who need to understand and modify the DSL program.

    In addition, comments can allow information to be passed to tools that generate structured documentation on the DSL programs. (Consider JavaDoc.)

19. *Provide organizational structures for DSL programs*

    To manage a DSL program that grows large and complex, we need to be able to break it up into subprograms. The subprograms may need to be organized into a graph structure and managed as a group in an archive.

    The DSL and the tools that manage DSL programs should allow a group of subprograms to be organized into "packages" that can be selectively included in other DSL programs.

    This solution seeks to manage complexity of DSL programs by breaking them into smaller subprograms. The "library" solution suggested for the "limit the number of language elements" guideline seeks to manage the complexity of the language itself.

20. *Balance compactness and comprehensibility.*

    Compact notation is generally efficient to write and process. However, it may not be comprehensible to the reader.

    Syntactic sugar can make a DSL more comprehensible, but make it more difficult to write. And, if overused, it can make the DSL confusingly verbose and thus less comprehensible.

    So we must balance among the factors. As noted above, we should normally favor the human reader over the writer.

21. *Use the same style everywhere.*

If a language has several sublanguages, make all the sublanguages similar in style. This makes the various languages easier to understand and use as a group.

For example, it would be confusing for one sublanguage to group items with `{` and `}` and another to use `begin` and `end` for a similar purpose.

Of course, this guideline must be balanced with the above guidelines suggesting we should "compose existing languages." "reuse existing language definitions," and "adapt existing notations domain experts use."

22. *Identify usage conventions.*

    To keep the DSL definition simple, we generally should not rigidly enforce minor syntactic issues such as layout. However, good DSL programming style can make a program more comprehensible (and more pleasant tor read).

    In parallel with the language definition, we should describe good style means for layout, naming conventions, order of elements, commenting, etc.

E. Abstract syntax guidelines (How to represent the DSL internally)

23. *Align abstract and concrete syntax*

    This means that:

    - elements that differ in concrete syntax should differ in internal representations
    - elements that are similar in *meaning* should have similar internal representations (e.g., be subclasses of same base class)
    - the internal representation of an element should not be dependent upon the context in which the element appears

    The goal is to be able to map the human readable representation of the DSL to the internal representation. This makes the execution easier to understand and debug. It may enable the generation of runtime error messages that tie back to the DSL program. (Remember the discussion of shallow versus deep embedding of DSLs.)

24. *Prefer layout that does not affect translation from concrete to abstract syntax.*

    This guideline suggests that issues like indentation should not affect the semantics. Not all language designers (e.g. of Python and Haskell) agree with this guideline.

25. *Enable modularity.*

To manage the complexity of the language, the "limit the number of language elements" guideline suggests providing a library of non-primitive elements that extend the core language.

To manage the complexity of large DSL programs, the "organizational structure for DSL programs" guideline suggests breaking a program into a group of related pieces and storing the pieces in a library of packages. This can be done at the DSL source code level.

This "enable modularity" guideline suggests the capability to build the internal representation of a DSL program by composing separately compiled "modules" incrementally.

Perhaps all of these mechanisms are implemented by a single module mechanism or there may separate mechanisms for composing precompiled modules, DSL source code packages, and DSL extensions.

26. *Introduce interfaces.*

    An *interface* usually defines a set of operation signatures (the name, parameters and their types, and return value type) and perhaps constraints on the operation's execution (preconditions and postconditions).

    If the modules of a large DSL program have well-defined interfaces, then the DSL's module mechanism can check whether the modules conform with each other's expectations.

    Interfaces and modules together support an information-hiding approach to program development. Each module "hides" its internal details from the other modules in the system.

More guidelines for internal DSL design

TODO: Better integrate the Karsai, Freeman, and other lists of guidelines.

Drawing on the experience in designing, implementing, and evolving the JMock internal DSL (which provides mock objects for testing), Freeman and Pryce make four recommendations for constructing an internal DSL in Java [Freeman 2006]. To some extent, these recommendations apply to design of all DSLs.

27. *Separate syntax and semantics (interpretation) into separate layers.*

    The concern of the syntax layer is to provide a fluent, readable language for users familiar with the application domain. These may not be programmers– or at least not programmers who are experts in the host language.

    The concern of the semantic layer is to provide a correct, efficient, and maintainable interpreter for the language. The developers and maintainers

of this layer are typically experts in the host language who can take advantage of the implementation language's capabilities and idioms.

As with most nontrivial software development tasks, mixing these concerns can make the implementation difficult to develop, understand, and maintain. It is better to translate the syntax to to an appropriate semantic model (e.g., an abstract syntax tree) for processing, hiding the details of each behind well-designed interfaces.

28. *Use, and perhaps abuse, the host language and its conventions to enable the writing of readable DSL programs.*

    For internal DSLs, the syntax layer may need to violate the conventional programming styles and naming conventions of the host language to achieve the desired readability and fluency.

    For example, DSLs in object-oriented languages may use *method chaining* and *method cascading* extensively to achieve the desired fluency. These practices usually discouraged in the usual programming practices.

29. *Don't trap the user in the internal DSL.*

    Note: This concern of this guideline is similar to discussion of libraries, packages, and modules in the Karsai et al guidelines in a previous subsection.

    The DSL should encapsulate its internal implementation details to avoid unexpected dependence on implementation features that might change over time and to enable naive users to use the DSL safely.

    However, it is difficult to anticipate all possible uses of a DSL over time. Thus it is helpful to enable expert users of the host language to extend the DSL by providing alternative implementations of key abstractions.

    The implementation of the DSL should itself be approached as a software family. Although the DSL syntax may look different than the host language, the DSL implementation should seek to work seamlessly with other host language programs.

    We should track the changes that expert users make. These help identify possible future enhancements of the DSL and its implementation.

30. *Map error reports to the syntax layer rather than to the semantics layer, which is hidden from the DSL user.*

    Note: The concern of this guideline is similar to that of Karsai et al's "align concrete and abstract syntax" guideline in a previous subsection.

    Good error reports are critical to a successful DSL. As much as possible, errors in both syntax and semantics should be stated in terms of the syntactic structure of the specific DSL program. This is often difficult to

accomplish, but it is important because it is unlikely that the DSL's users will be familiar with the internal details of the implementation.

Deep embedding of DSLs can make it easier to trace errors back to recognizable syntactic structures.

Conclusion

TODO

Exercises

TODO

Acknowledgements

In Fall 2016, I adapted and revised much of this work for possible use in CSci 450 (Organization of Programming Languages), but I did not use it that semester. These notes are based, in part, on a previous HTML-source handout on domain-specific languages for exercises in the Fall 2014 CSci 450 and Spring 2016 CSci 555 classes. The notes draw ideas from several of the references listed in the final section.

For a Haskell-based offering of CSci 556 (Multiparadigm Programming) in Spring 2017, I continued to develop these notes, adding discussion of the boundaries between DSLs and other computing artifacts, of the Expression Language case study, and of my use of DSLs in the workflow for producing this document.

I updated these notes slightly in Summer and Fall 2017 for possible use in CSci 450, but did not use it that semester.

In Spring 2018, I added discussion of Fowler's definition of DSLs, advantages and disadvantages of DSLs, and DSL design guidelines (from Karsai et al) and modified the document for use in CSci 658 (Software Language Engineering).

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed. The HTML version of this document may require use of a browser that supports the display of MathML.

References

[**Bentley 1986**] J. Bentley. Programming Pearls: Little Languages, *Communications of the ACM*, Vol. 29, No. 8, pp. 711-721, August 1986. [local]

[**Brooks 1987**] Frederick P. Brooks. No Silver Bullet: Essence and Accident in Software Engineering, *IEEE Computer*, Vol. 20, No. 4, 10-19, 1987.

[**Coplien 1998**] J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering, *IEEE Software*, 15(6):37–45, November 1998. [local]

[**Cunningham 2008**] H. C. Cunningham. A Little Language for Surveys: Constructing an Internal DSL in Ruby, In *Proceedings of the ACM SouthEast Conference*, 6 pages, March 2008.

[**Fowler 2008a**] M. Fowler. DomainSpecificLanguage, Blog posting, 15 May 2008 (accessed 16 January 2018).

[**Fowler 2008b**] Martin Fowler. One Lair and Twenty Ruby DSLs, Chapter 3, *The ThoughtWorks Anthology: Essays on Software Technology and Innovation*, The Pragmatic Bookshelf, 2008. [local chapter]

[**Fowler 2011**] M. Fowler. *Domain Specific Languages*, Addison Wesley, 2011.

[**Freeman 2006**] S. Freeman and N. Pryce. Evolving an Embedded Domain-Specific Language in Java, In *Companion to the Conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 855–865. ACM SIGPLAN, October 2006. [local]

[**Hudak 1996**] Paul Hudak. Building Domain-Specific Embedded Languages, *ACM Computing Surveys*, Vol. 28, No. 4, p. 196, 1996. [local]

[**Hudak 1998**] P. Hudak. Modular Domain Specific Languages and Tools, In P. Devanbu and J. Poulin, editors, *Proceeding of the 5th International Conference on Software Reuse (ICSR'98)*, pages 134-142. IEEE, 1998. [local]

[**Karsai 2009**] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler and Steven Voelkel. Design Guidelines for Domain Specific Languages, In *Proceedings of OOPSLA Workshop on Domain-Specific Modeling*, 2009. Also arXiv preprint arXiv:1409.2378, 2014. [local]

[**MacFarlane 2018**] J. MacFarlane and the Pandoc community. Pandoc: A Universal Document Converter Wht, accessed 17 January 2016.

[**Mernik 2005**] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain Specific Languages, *ACM Computing Surveys*, 37(4):316-344, December 2005. [local]

[**Spinellis 2001**] Diomidis Spinellis. Notable Design Patterns for Domain-Specific Languages, *Journal of Systems and Software*, Vol. 56, No. 1, pp. 91-99, 2001. [local]]

[**Thibault 1999**] S. Thibault, R. Marlet, and C. Consel. Domain-Specific Languages: From Design to Implementation–Application to Video Device Driver Generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May/June 1999.

[**Thompson 2011**] S. Thompson. *Haskell: The Craft of Functional Programming*, Third Edition, Addison Wesley, 2011.

[**van Deursen 2000**] A. van Deursen, P. Klint, and J. Visser. Domain Specific Languages: An Annotated Bibliography, *SIGPLAN Notices*, 35(6):26-36, June 2000. [local]

[**Ward 199**] M. P. Ward. Language-Oriented Programming, *Software–Concepts*

and Tools, Vol. 15, No. 4, pp. 147-161, 1994. [local]
[**Wikipedia**] Wikipedia. Domain-Specific Language, accessed 12 April 2017.

Concepts

TODO: Update this list

Domain-specific languages (DSLs); language nature; fluency; limited expressiveness; domain focus; DSLs versus general-purpose programming languages; DSLs versus APIs; external versus internal DSLs; shallow versus deep embedding of internal DSLs; use of algebraic data types to implement DSLs