Section 1: Week 3: Domain Specific Languages

Nate Bachmeier

TIM-8110: Programming Languages and Algorithms

May 19, 2019

North Central University

# Domain Specific Languages

A Domain Specific Language (DSL) is a mechanism for concisely describing interactions within a well-defined domain. Even the most novice of computer programmer uses dozens of these languages, perhaps without knowing,

A General-Purpose Languages (GPL) is a mechanism for describing problems that span multiple application domains. They tend to be more verbose than DSL due to needing to specify both the domain context and the interaction.

# Web Application Example

To build a simple webpage requires source files annotated with HTML and CSS. These languages describe how the content should be structured and presented. To publish updates the deployment team kicks off a shell script. Within that deployment script are Regular Expressions (RegEx) to transform place holders into finalized values within the configuration files. Each of these steps been composed within DSL (Saltmarch, 2013).

Next the server-side code uses the ASP.net framework which is written in C#. To describe the data binding scenarios, a mash up of XML and C# called Razor is often used.

Both C# and XML are GPL languages as they are used across a wide range of software domains such as client applications, automotive systems, and data science pipelines. The Razor templates are another example of DSL as they solely exist within the domain of ASP.net data binding. It does not matter that the Razor templates are using C# for their implementation syntax. The key distinction is the contextually specific use case.

# Categories of DSL

Domain Specific Languages can be categorized into distinct groups markup, modeling, and programming (Ghosh, 2011). These categories can be sub-divided into internal and external languages.

## Markup

Markup languages such as HTML, LaTex, and Markdown add metadata – such as formatting and font sizes within a textual document. Another common scenario is place attributes across the code base and then drive runtime or compile decisions through metaprogramming.

## Modeling

Modeling languages describe an object hierarchy and their relationships. XML and JSON configuration files are common methods for persisting the model’s representation. It is acceptable to implement concrete DSLs within these abstract GPL, though there are some draw backs. Most notably that the syntax is fixed and cannot be easily extended to add freeform expressiveness.

This limitation can be mitigated by constructing micro languages through grammar file tooling such as JavaCC or ANTLR.

Perhaps a family ancestry application uses this approach to expose the command ADD Jared AS BROTHER instead of <Add> <Relationship Type=”Brother”> <Name>Jared</Name> </Relationship> </Add>. The first scenario is more compact and easier for a novice to understand. The second requires less custom code to parse into an abstract representation and begin traversing.

The ancestry application could also expose rich modeling syntax for traversing the lineage. Consider the command to find a person’s male first cousins once removed as  
(me) > (parent) > (cousin[gender:male]).

This example shows how DSL improves readability by making the intent clear, and the code is nearly self-documenting. The DSL query model is also sufficiently intuitive that a layman could discover similar operations such as their mother’s uncles (me) > (parent[gender:female) > (uncle).

## Domain Scripting

Domain Programming languages extend modeling languages to include branch and loop control structures. TradeStation’s EasyLanguage allows business users to automate stock trading strategies. An end user might purchase shares of Apple with the statement IF current\_price < desired\_price THEN BUY 100 SHARES OF APPLE AT MARKTET PRICE (Ruggiero, Brower, & Putt, 2015).

The intent of that statement is instantly understandable to both the programmer and the domain expert. This clarity allows those experts to become more deeply integrated into the development cycle and ensure business rules are properly implemented. Through decoupled responsibilities, it become possible for specialists to operate on each aspect of the problem.

# Internal vs External

An internal DSL is embedded within the context of a host GPL. An external DSL’s source files are consumed and transformed through separate tooling.

## Simplicity vs Customizability

Internal languages often use creative tricks to improve the readability within their host language such as operator overloading, removing optional punctuation, and defining no/op bubble words. These internal languages can be parsed along with the other source code and compiled in a similar manner.

The proposed ancestry query language could have implemented in C++ by overloading the GreaterThan and Index operators. However, there are limits to this approach.

Perhaps the business unit identifies a value-add by introducing an operator for querying great grandparents. Based on usability studies it is decided that prefixing the selector (>) with one equal sign per generation is the most popular. For instance, the 3rd generation would be accessed as ===>. Thus (me) > (parent) > (grandparent) > (great grandparent) > (cousins) is equal to (me) ===> (cousin).

It would not be possible to implement this design in many host languages as they do not expose this operator in the first place. To gain additional flexibility the development team needs to use an external DSL language, and parse the commands into an abstract representation. This flexibility comes at the cost of being more effort to maintain the custom grammar files.

# Criticisms

## Complexity to Learn

A common criticism of DSL is that it increases the system complexity due to the development team needing to understand dozens of micro languages. The alternative is to create an Application Programming Interface (API) and require the development team to understand that instead.

If the DSL is properly designed, then it should be less than or equal to the complexity of the API (Fowler, 2006). This is because the DSL (1) does not need to solve general problem, (2) can focus on the specific task at hand, (3) and can clearly express the intent through keywords and dedicated operators.

## Lack of Tooling and Third-Party Libraries

Another criticism of external DSL is that they lack the tooling and third-party libraries that are available to both internal DSL and GPL. For example, an internal DSL implemented in Ruby can easily import a Gem and perform any custom action. The authoring experience can also take place within an Integrated Developer Environments (IDE), which offers debugging auto completed statement scenarios.

One solution is to expose grammar for binding into Shared Objects (SO) and Dynamic Link Libraries (DLL). For example, Easy Language supports the command structure EXTERN MyFunc(String,String) FROM MyLib.dll. Users can then use the MyFunc delegate as a mechanism to bridge the control flow into native components.

Perhaps MyLib.dll contains highly optimized physics functions written in Fortran, and the cost is prohibitively expensive to rewrite them. Even if cost was not an issue, it might be unlikely the code would be more efficient after being ported.

The language authors need to be mindful that these extensions do not push the system into being a full feature GPL. With each step in that direction the separation of responsibilities between domain expert and system engineer fades. Eventually the scripts and configuration can only be written and maintained by development staff, which greatly reduces the key benefit of DSL-- limiting required context to a single domain.

# Putting it all Together Attack Scripting Example

Security engineers often write their exploit validation scripts in Nessus Attack Script Language (Nasl), Perl, or Python. Nasl is an external DSL with grammar for opening ports and modifying binary structures. Metasploit offers an internal DSL through a collection of Perl extensions. Python is a GPL with an enormous module library.

Nasl is easy to read and write due to the removal of boiler plate code and the contextually specific commands. Metasploit must live within the Perl syntax which can lead to non-intuitive argument passing in some scenarios. Python requires the most knowledge as each script manages the lifecycle of the module explicitly.

Full development environments exist for all three languages, though Perl and Python clearly have better support. Their larger communities invite more investments which leads to better tooling around the code artifacts. For instance, there are multiple implementations of ‘perldoc’ but only one official documentation solution for Nasl. This leaves the developers with many ‘take it or leave it’ decisions; as it is often hard to justify building additional custom tooling.

Ideally the developer can author in the language that is the best tool for the task at hand. Perhaps a utility could be created for compiling Nasl into C/C++ code. This could be imported into either Perl or Python environments. This would lower the learning curve as simple Nasl commands could expand out to very sophisticated network operations. Alternatively, maybe inline Python within the Nasl script covers any deficiencies. For any of the simple common scenarios, the goal should be to keep them simple.

# Conclusions

Domain Specific Languages allow the user to declare interactions within a specific application domain. General-Purpose Languages need to work across a wide range of scenarios and platforms which prevents them from making certain assumptions. These contextually sensitive assumptions remove boiler plate code and improves the readability of the language code.

DSL languages appear in markup, modeling, and domain scripting scenarios. Their use cases support the decoupling of domain experts and system engineers. The domain expert can then be more integrated into the development process and catch contextual fallacies early on. This will save both time and resources for the business.

DLS languages can be implemented as either internal or external. Internal languages are easier to build as they reside within the syntax of the host language. However, there are limitations as the GPL enforces its syntactical rules. Instead external DSL can be used to introduce custom parser grammar for consuming arbitrary code and configuration data.

Ultimately DSL and GPL languages allow the system designer the flexibility to model different aspects of the system in a manner than is most readable and maintainable. Aspects of the system that are unlikely to change can be authored in GPL. Meanwhile business rules and configuration should be pushed into DSL languages. This allows for domain experts to audit the rules and provide guidance without needing an advanced degree in computer science.

# References

Fowler, M. (2006, October 31). *An Introduciton to Language Oriented Programming.* Retrieved from InfoQ: https://www.infoq.com/presentations/domain-specific-languages

Ghosh, D. (2011). DSL for the Uninitiated. *Communications of the ACM Vol. 54 No. 7*. Retrieved from queue.acm.org.

Ruggiero, M., Brower, W., & Putt, A. (2015). *Using EasyLanguage 9.x.*

Saltmarch. (2013, February 22). *Harnessing Domain-Specific Languages (DSLs).* Retrieved from YouTube: https://youtu.be/AdMxlTzjARw