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. Perhaps without knowing, even the most novice of computer programmer uses dozens of these languages.

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 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.

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# as their implementation syntax. The key distinction here is the contextual use case.

# Categories of DSL

Domain Specific Languages can be categorized into distinct groups markup, modeling, and programming. 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 of persisting their 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 expressiveness.

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

Perhaps an 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 effort 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 (me) > (parent) > (cousin[gender:male]) limit 10. This query finds up to 10 of my first cousins once removed. The readability of this statement within the context is sufficiently high that no additional details are needed.

## Domain Scripting

Domain Programming languages extend modeling languages to also add control flow such as branching and loops. TradeStation’s EasyLanguage allows business users to automate stock trading strategies. These users can specify IF current\_price < desired\_price THEN BUY 100 SHARES OF APPLE AT MARKTET PRICE.

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. That can reduce the costs to implement new features responsibilities are decoupled and specialists operate on each aspect of the problem.

# Internal vs External

An internal domain specific language is embedded within the context of its parent general purpose language, while an external resides source files that are consumed through additional tooling.

## Simplicity vs Customizability

Internal languages often use creative tricks to improve the readability of their language such as operator overloading, removing optional punctuation, and defining no/op bubble words. The proposed ancestry query language could have implemented in C++ by overloading the GreaterThan and IndexInto operators.

There are limits to this approach and challenges to internal languages. The business unit notices that requesting great grandparents appears in a large set of queries. They want to allow specifying the hierarchical levels as a sequence of equal signs, such that (me) > (parent) > (grandparent) > (great grandparent) > (cousins) is equal to (me) ===> (cousin). If the internal language does not expose an ===> then it cannot be overridden, and the feature cannot be implemented.

To gain additional flexibility the development team needs to use define an external DSL language and parse the commands into an abstract representation. This flexibility comes at the cost of being more effort to maintain custom grammar files.

# Criticisms

## Complexity to Learn

Common criticism of external DSL is having to learn dozens of micro languages adds to the complexity of the system. If the domain problem was instead embedded in the GPL, users would have to learn the Application Programming Interface (API). The DSL can reduce that learning curve by offering syntactical sugar and focusing on narrow slivers of the design.

## Lack of Tooling and Third-Party Libraries

Another criticism of external DSL is that they lack the tooling and rich library support that is available to both internal DSL and GPL. For example, an internal DSL implemented in Ruby can easily import a module to perform any custom action. The internal DSL also comes with existing Integrated Developer Environments (IDE), which are already part of the development workflow.

One solution to this problem is to expose grammar for binding 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 for bridging the communication to MyLib.dll. Perhaps that library contains highly optimized physical functions written in Fortran, and the cost is prohibitively too high to simply rewrite and test them. Even if that was not the case, it is unlikely they would be faster or more efficient.

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

# Concrete Example with Attack Scripting

Security engineers often write their exploit validation scripts in Nessus Attack Script Language (Nasl), Perl, or Python. Nasl is an external domain language specific language with grammar for opening ports and easily modifying binary structures. Metasploit offers an internal domain specific language through a collection of Perl extensions. Python is a general-purpose language with a rich module library.

Nasl is the very easy to read or write due to the removal of boiler plate code and the contextual driven 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 it must manage the lifecycle of the module explicitly.

Full developer environments exist for all three languages, though Perl and Python clearly have better support. Among these larger communities comes better tooling for the lifecycle of the code artifacts. For instance, there are multiple implementations of perldoc but only one documentation system for Nasl. This leaves the developers with a “take it or leave it” decision as it is often hard to justify investing in more 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.

# Conclusions

Domain Specific Languages allow the user to declare interactions within a specific application context. 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. These 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.

Each category of DSL language can be implemented as an internal or external language. 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 use grammar files to consume 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 the parts 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.