Domain Specific Language – A Solution to the Intricacies of General Purpose Language

Sandeep. R Govind. S Goutham. V Atreya Rekha. P. M

Abstract— Domain Specific Language (DSL) has been in the use from a long time. However, DSLs came to the limelight only in the recent past, around the time Java gained importance. The reason for this sudden upsurge in DSLs, the advantages it has to offer over General Purpose Languages (GPL), the inherent complexities involved in the development of the language are discussed in the paper. Although a number of tools exist, this paper briefs on the use of ANTLR, for the development of a DSL.

Keywords — ANother Tool for Language Recognition (ANTLR), Domain Specific Language (DSL), Domain Specific Modeling (DSM)

I. INTRODUCTION

DSL, Domain Specific Language is a language that is tailored to a specific application domain [1]. Domain-specific modeling (DSM) allows domain-experts to play active roles in development efforts. The idea of domain specificity is neither new nor unheard of. Biologists and other scientists' use parsers for pattern recognition and analysis. There are certain approaches in all branches of Engineering and Science that are generic and those that are specific [2]. A generic approach applies for a vast area of problems, but such a solution may be suboptimal. A specific approach provides a much better solution for a focused, smaller set of problems. Domain specific modelling (DSM) enables experts of arbitrary domains to perform modelling tasks using familiar constructs. Domain specific concepts and their relationships are captured by domain-specific languages (DSLs).

Domain Specific Languages are highly expressive and easy to comprehend than the General Purpose Languages (GPL). However, the effort required to develop a DSL is enormous, as it requires the thorough understanding of the problem domain and also the complexities involved in the process of language development.

DSLs improve productivity for developers and enhance the communication with domain experts [3]. The largest implementations of DSLs can be seen in business establishments, as a tool to express and implement the business rules, due to the fact that DSLs trade generality for expressiveness in a limited domain [1].

Sandeep R, Govind S and Goutham V Atreya are pursuing 8th semester, Bachelor of Engineering, in the Information Science and Engineering (ISE) department at JSS Academy of Technical Education, Bangalore.

Rekha P M is a faculty, department of Information Science and Engineering, JSS Academy of Technical Education Bangalore.

A thorough understanding of the problem domain is a prerequisite for developing the DSL. The analysis of the application domain is provided by domain analysis [4]. The results of this are obtained in a feature model [5]. Often, the target for code implementing a DSL will itself be a simple, abstract machine [6].

ANTLR framework provides a parser. It can automatically build parse trees, and generate tree walkers that help analyze and execute the application specific code [7].

II. IMPLEMENTATION AND DSL LIFE CYCLE

A DSL behaves as a tool with a restricted focus, only for a specific domain. Unlike object orientation or agile processes, which focus on the primary shift into the way software is developed [3]. A DSL should define both syntax and semantics of its domain of application. In particular, a DSL may be an orthogonal extension to UML instead of only a restriction of existing UML concepts [8].

DSL is a thin veneer over a model where a model might be a library or a framework (here ANTLR). DSL clearly appeals to communicate the intent of a part of system. Easier it is to read a lump of code, easier it is to find the mistakes and easier it is to modify the system [9].

A DSL enhances the understanding of how to use an API since it shifts focus to how different API methods should be combined together. The usefulness of a DSL is that it provides an abstraction that one can use to think about a subject area [10]. The behavior of the domain is expressed more easily than in terms of lower level constructs.

An internal DSL is pattern of writing code in the host language. Example: A C# internal DSL is C# code written in particular style for giving a more language-like feel. They are alternately termed as Fluent Interfaces or Embedded DSLs [9].

An external DSL is a completely separate language that is parsed into data that the host language can understand [9].

A. DSL Lifecycle

Primarily, it's a better approach to define the DSL. It begins with some scenarios and the way it is conceptualized. This serves as the communication medium [11]. The beginning might be with statements that are to be syntactically correct. For internal DSL, adherence to the syntax of the host language is necessary. For external DSL, statements are written which can be parsed in a way the host language understands. Later, the state machine is built, at this point the interaction with the customers happen in order to clearly understand the needs. This terminates with the establishment of a set of example controller behaviors [12].

For each case identified earlier, a DSL is tried to develop in some form. As it progresses, modifications to DSL is done to support new capabilities. By the end, a reasonable sample of cases and a pseudo DSL description of each would be developed.

With the pseudo DSL, implementation is begun [13]. Implementation phase can be done in many ways - Designing the state machine model in host language, or a command query API for the model is written, concrete syntax of DSL, or translation between DSL and command query API.

III. DSL Processing – Architecture

The architecture of DSL processing, as shown in Fig. 1 [3], is described by capturing the actual theme in a model and the DSL's role is to populate the model via a parser. A Semantic model can be an object model that combine data and processing, but it need not necessarily be so, it can also just be a Data Structure [6].

Domain model is used to capture the core behavior of the software system. However, semantic model of a DSL is the subset of the application's domain model.

At this point the major difference between the internal DSL and external DSL is visible - the difference lies in the parsing step, both in what is parsed and in how the parsing is done [1]. However both styles of DSL will produce the same lines of semantic model.

In External DSL, there is transparency between DSL scripts, parser, and semantic model. While with internal DSL, it gets jumbled .Thus in an internal DSL, paring DSL scripts is done by a combination of host language parser and expression building.

After building the model, we need to make the model perform the task needed. This can be done in two ways. The simplest and the best way is to execute the model [12]. The semantic model is the code and as such it can run and do what it needs to do. The next option is to use code generation; here code is separately compiled and run. Yet DSLs have no inherent need for code generation. A lot of the time the best thing to do is just to execute the Semantic Model.

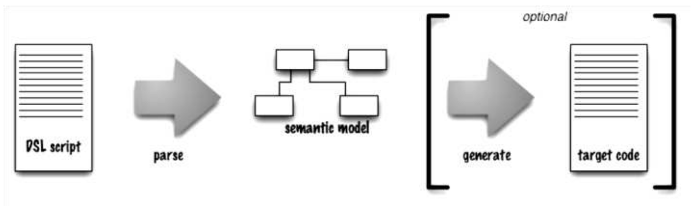


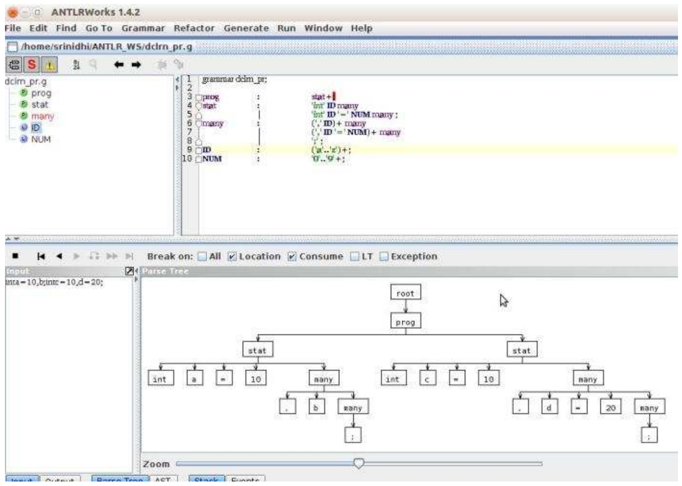
Fig. 1 Overall Architecture of DSL Processing

IV. ANTLR

ANTLR is basically used by the programmers as, a parser generator to build translators and interpreters for DSLs. DSLs include data formats, configuration file formats, network protocols, text-processing languages, protein patterns, gene sequences, space probe control languages, and domain-specific programming languages [7].

A. Implementing Abstract Syntax Trees

ANTLR assumes nothing about the actual Java type, the implementation of AST nodes and tree structure but requires that developer specifies or implement a TreeAdaptor [7]. It is often the case that programmers either have existing tree definitions or need a special physical structure, thus, preventing ANTLR from specifically defining the implementation of AST nodes. ANTLR specifies only an interface describing minimum behavior. The tree implementation must implement this interface so ANTLR knows how to work with these trees. Further, developer must tell the parser the name of his tree nodes or provide a tree "factory" so that ANTLR knows how to create nodes with the correct type (rather than hard coding in a `new AST()` expression everywhere). ANTLR can construct and walk any tree that satisfies the AST interface. A number of common tree definitions are provided. Unfortunately, ANTLR cannot parse XML DOM trees as method names conflict (e.g., `getFirstChild()`) [10].


ANTLRWorks 1.4.2
File Edit Find Go To Grammar Refactor Generate Run Window Help
/home/srinidhi/ANTLR_WS/dclrn_pr.g
dclrn_pr.g
prog
stat
many
ID
NUM
Break on: All Location Consume LT Exception
Input
Parse Tree
root
prog
stat
int
a
=
10
many
,
b
many
;
stat
int
c
=
10
many
,
d
=
20
many
;
Zoom

```
grammar dclrn_pr;

prog        :        stat+
stat        :        'int' ID many
            |        'int' ID '=' NUM many ;
many        :        (',' ID)+ many
            |        (',' ID '=' NUM)+ many
            |        ';' ;
ID          :        ('a'..'z')+;
NUM         :        '0'..'9' +;
```

```
inta=10,b;intc=10,d=20;
```

Fig. 2 Parse tree for syntax checking grammar.

Fig. 2 is a snapshot of parse tree for a syntax checking grammar, taken from ANTLRWorks Grammar Development Environment [7], for the input stream

```
int a =10,b;
int c=10,d=20;
```

and the grammar,

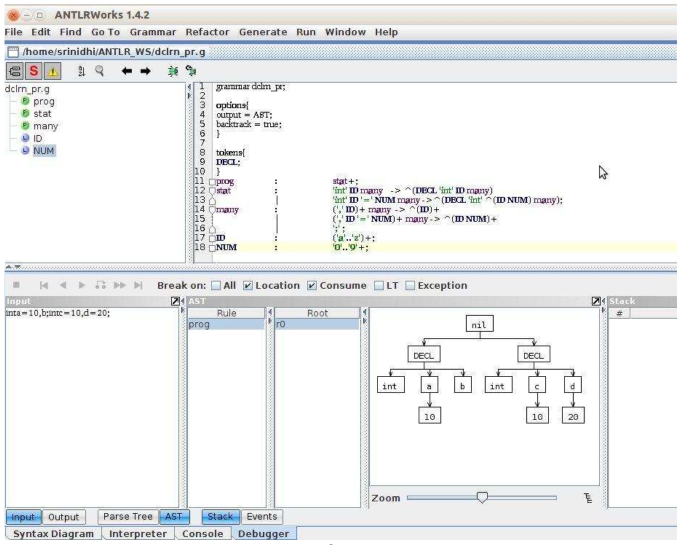
```
grammar dclrn_pr;

prog   :   stat+
       ;
stat   :   'int' ID many
       |   'int' ID '=' NUM many
       ;
many   :   (',' ID)+ many
       |   (',' ID '=' NUM)+ many
       |   SEMI
       ;
```

```
SEMI  :   ';'
      ;
ID    :   ('a'..'z')+
      ;
NUM   :   '0'..'9'+
      ;
```

Note that the grammar takes care of the syntax and not the semantics.

The recommended way to build ASTs is to add rewrite rules to the grammar. Rewrite rules are like output alternatives that specify the grammatical, two-dimensional structure of the tree one has to build from the input tokens [10]. The notation is as follows:

```
rule: «alt1 » -> «build-this-from-alt1 »
    | «alt2 » -> «build-this-from-alt2 »
      ...
    | «altN » -> «build-this-from-altN »
    ;
```


ANTLRWorks 1.4.2
File Edit Find Go To Grammar Refactor Generate Run Window Help
/home/srinidhi/ANTLR_WS/dclrn_pr.g
dclrn_pr.g
prog
stat
many
ID
NUM
Break on: All Location Consume LT Exception
Input
AST
Rule
Root
Stack
nil
DECL
DECL
int
a
b
int
c
d
10
10
20
Zoom
Input Output Parse Tree AST Stack Events
Syntax Diagram Interpreter Console Debugger

```
grammar dclrn_pr;

options{
output = AST;
backtrack = true;
}

tokens{
DECL;
}
prog    :    stat+;
stat    :    'int' ID many   -> ^(DECL 'int' ID many)
        |    'int' ID '=' NUM many -> ^(DECL 'int' ^(ID NUM) many);
many    :    (',' ID)+ many -> ^(ID)+
        |    (',' ID '=' NUM)+ many -> ^(ID NUM)+
             ';';
ID      :    ('a'..'z')+;
NUM     :    '0'..'9'+;
```

```
inta=10,b;intc=10,d=20;
```

Fig. 3 Abstract Syntax tree for syntax checking grammar.

Fig. 3 depicts the snapshot of the Abstract Syntax Tree for the same syntax checking grammar of Fig. 2. Careful observation shows the rewriting of the rules (in italics). The options block tells ANTLRWorks to output AST along with Parse tree, rather than Parse tree alone. The rewritten grammar is,

```
grammar dclrn_pr;
options{
output = AST;
backtrack = true;
}
tokens{
DECL;
}

prog   :   stat+
       ;
stat   :   'int' ID many           -> ^(DECL 'int' ID many)
       |'int' ID '=' NUM many -> ^(DECL 'int' ^(ID NUM) many)
       ;
many  :   (',' ID)+ many                -> ^(ID)+
      |   (',' ID '=' NUM)+ many      -> ^(ID NUM)+
      |   SEMI
      ;
SEMI  :   ';'
      ;
ID    :   ('a'..'z')+
      ;
NUM   :   '0'..'9'+
      ;
```

B. Generating Structured Text with Templates and Grammars

StringTemplate is a java template engine (with ports for C#, Python) for generating source code, web pages, emails, or any other formatted text output. StringTemplate is particularly good at code generators, multiple site skins, and internationalization/localization[10]. Each rule's template specification creates a template using the following syntax (for the common case):

... -> template-name(«attribute-assignment-list »)

The resulting recognizer looks up template-name in the StringTemplate. StringTemplateGroup is a group of templates that acts like a dictionary, which maps template names to template definitions. The attribute assignment list is the interface between the parsing element values and the attributes used by the template. ANTLR translates the assignment list to a series of setAttribute() calls[7].

C. Bytecode Interpreter

The executable instructions for the translator are as depicted in the Table I [7] [9].

TABLE I
JAVA BYTECODE NEEDED FOR EXPRESSION

BYTECODE INSTRUCTION	Description
ldc	Integer-constant Push constant onto stack.
imul	Multiply top two integers on stack and leave result on the stack. Stack depth is one less than before the instruction. Executes push (pop*pop).
iadd	Add top two integers on stack and leave result on the stack. Stack depth is one less than before the instruction. Executes push (pop+pop).
isub	Subtract top two integers on stack and leave result on the stack. Stack depth is one less than before the instruction. Executes b=pop; a=pop; push (a-b).
istore local-var-num	Store top of stack in local variable and pop that element off the stack. Stack depth is one less than before the instruction.
iload local-var-num	Push local variable onto stack. Stack depth is one more than before the instruction.

V. Background and Related Work

The use of DSLs for formally specifying a member of a system family is outlined by Czarnecki and Eisenecker [14]. The authors call such a DSL a configuration DSL and the language can be derived from the feature model [15]. The most popular means of DSL abstract syntax specification and communication today are UML class diagrams and human readable textual notations (HUTNs) [16].

Recent parallel programming languages include Chapel [17], Fortress [18], and X10 [19]. These languages employ explicit control over locations and concurrency and are targeted primarily at scientific applications for supercomputers [20].

A declarative data description language, PADS - Processing Ad hoc Data Sources, allow data analysts to describe both the physical layout of ad hoc data sources and semantic properties of that data. Using these descriptions, libraries and tools for manipulating the data, including parsing routines, statistical profiling tools, translation programs to produce well-behaved formats such as XML or those required for loading relational databases, and tools for running XQueries over raw PADS data sources, are generated by the PADS compiler [21].

VI. Conclusions

At the turn of the millennium, even with the standardization of programming languages, a wide gap existed – bits of important logic were not fitting into the constructs of even the best of languages. This possibly led to the DSL revolution.

Domain specific languages hold the promise of delivering high payoffs in terms of software reuse, automatic program analysis, and software engineering. The ever changing business standards and business environment profited from the tailor made languages that could bridge the business mind with that of the developer.

DSLs are here to stay for a longer time until a more powerful motivation paves the way for a novel alternative altogether.

Acknowledgment

The authors thank Dr. D V Ashoka, Professor and Head, Department of Information Science and Engineering, JSS Academy of Technical Education, Bangalore, Mrs. Sneha Girish, faculty, department of Computer Science and Engineering, JSSATE Bangalore for their constant review and support in writing this paper and Mr. Madhukara Phatak, Alumni JSSATE Bangalore, for helping learn ANTLR.

References

[1] ACM Computing Surveys, Vol. 37, No. 4, December 2005, pp. 316–344.
[2] A. V. Deursen, P. Klint, J. Visser, Domain-Specific Languages: An Annotated Bibliography, Dutch Telematica Instituut.
[3] M. Fowler, Domain Specific languages, Addison-Wesley Professional, 2010, ISBN 0-321-71294-3 .
[4] I. Fister Jr. et al., Computer Languages, Systems & Structures 37, pages 151–167, 2011.
[5] Schobbens P-Y, Heymans P, Trigaux J-C, Bontemps Y., Generic semantics of feature diagrams - Computer Networks, pages 456–79, 2007.
[6] Richard B. Kieburtz, Defining and Implementing Closed, DomainSpecific Languages, Oregon Graduate Institute of Science & Technology .
[7] Terence Parr, The Definitive ANTLR Reference Building Domain-Specific Languages, The Pragmatic Bookshelf, ISBN-10: 0-9787392-5-6, 2007.
[8] Achim D. Brucker, Jurgen Doser, Metamodel-based UML Notations for Domain-specific Languages, Information Security, eth Zurich, 8092 Zurich, Switzerland.
[9] S. Thibault, R. Marlet, C. Consel, Domain-Specific Languages: from Design to Implementation Application to Video Device Drivers Generation, IRISA / INRIA - Universit´e de Rennes 1 Campus Universitaire de Beaulieu 35042 Rennes cedex, France.
[10] Terence Parr, Language Implementation Patterns – Create your own Domain Specific and General Programming Language, The Pragmatic Bookshelf , ISBN-10: 1-934356-45-X.
[11] Thibault S, Marlet R, Consel C., Domain-specific languages: from design to implementation—application to video device drivers generation, IEEE Transactions on Software Engineering 1999;25(3), pages 363–77.
[12] Sprinkle J, Mernik M, Tolvanen J-P, Spinellis D., What kinds of nails need a domain-specific hammer?, IEEE Software 2009;26(4).
[13] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools,Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition, 2006.
[14] Czarnecki, K., Eisenecker, U.W, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[15] Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.
[16] James R. Cordy, The txl source transformation language. Science of Computer Programming, 61:190–210, 2006.
[17] B. Chamberlain, D. Callahan, and H. Zima, Parallel Programmability and the Chapel Language, Int. J. High Perform. Comput. Appl., vol. 21, no. 3, pp. 291–312, 2007.
[18] G. L. S. Jr., Parallel programming and parallel abstractions in fortress, in IEEE PACT, 2005, p. 157.
[19] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, X10: an object-oriented approach to non-uniform cluster computing, SIGPLAN Not., vol. 40, no. 10, pp. 519–538, 2005.
[20] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, A Heterogeneous Parallel Framework for Domain-Specific Languages, 2011 International Conference on Parallel Architectures and Compilation Techniques.
[21] Robert Gruber, Kathleen Fisher,PADS: A domain specific language for processing ad hoc data, http://www.padsproj.org/doc.html#papers.