Domain-Specific Languages: A Theoretical Survey

Nuno Oliveira[1], Maria João Varanda Pereira[2], Pedro Rangel Henriques[1], and Daniela da Cruz[1]

[1] University of Minho - Department of Computer Science,
Campus de Gualtar, 4715-057, Braga, Portugal
{nunooliveira, prh, danieladacruz}@di.uminho.pt
[2] Polytechnic Institute of Bragança
Campus de Sta. Apolónia, Apartado 134 - 5301-857, Bragança, Portugal
mjoao@ipb.pt

Abstract. Domain-Specific Languages (DSLs) are characterized by a set of attributes that make them different and easy to use when compared to General-purpose Programming Languages (GPLs). The fact of being tailored for a specific domain rises many advantages on their usage, however special care must be put in their conception and implementation.
The purpose of this paper is to provide a survey on DSLs, enhancing their characteristics that make clear the advantages and disadvantages of their usage and make challenging their implementation. We also focus on the development methodologies that have been used to create the thousands of DSLs that exist today, which are a powerful alternative to GPLs.

1 Introduction

No matter what is done, computers will always understand things in terms of 0's (zeros) and 1's (ones). These *things* they understand are human-made programs. Humans, although capable of doing it, are not proficient in *speaking* that binary language. However, the computer can be taught to translate any language into its preferred idiom. For this reason, humans do not need to go down to binary level. Instead, they can keep the way they talk to computers at a very perceptible level by rising the abstraction level of the language they use.

Programming is a computer-oriented task, but other actors are also involved on it. One of the most important concerns when developing a software piece is about its future, namely, its maintenance. It is not a computer that will maintain the software, but a human; so humans must be taken into consideration when choosing the programming language and writing the code. Thus, the terms and concepts used in the programming language (its syntax and semantics) should be close to those persons that have to maintain the software.

Two types of languages are mainly used to write programs for computers: GPLs and DSLs.

There is not a precise definition for GPL [1]. GPLs are tailored to be used to solve any kind of problem, no matter the area or domain this problem fits into. Normally they are the programmer's preference for the communication with the computer. Many relevant factors contribute to this choice. As they are general purpose and widely used,

these languages are adopted by the majority of programmers. The existence of a large community of experts in one GPL also plays an important role in such adoption. But issues like the maturity of a language the availability of well tested optimizing compilers, and the existence of good development tools or environments [2], are the more crucial for the referred preference. But on the other hand, GPLs also have many drawbacks when regarding other aspects like writing and reading (the learning curve has a long setup time), or understanding their programs. Concerning the former aspect (writing and reading), they always imply vast programming expertise. Hence, not every one is able to use them properly. Concerning the latter aspect (program comprehension), GPLs are hard to understand because on the one hand, their syntax and semantics is not obvious due to their generality, and on the other hand they commonly address implementation particularities that are closer to the machine's way of working than to the human's way of thinking. Even following different programming paradigms or exhibiting various syntactic constructions, which aims at raising the abstraction level, is not enough to overcome the difficulties observed on comprehending GPL programs.

DSLs [3] can be defined by the following sentence:

> *A domain-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.* [4]

As these languages are conceived for a specific domain, it is easy to absorb the main concepts and features inherent to this domain, and bring them to the language constructors [5]. This should result in a syntax and semantics capable of effectively raising the abstraction level of the programming [6] task; that is, the notation addresses a higher level of abstractness, increasing the distance to the machine's way of working and shortening it to the human's way of thinking. Some examples of DSLs include languages like LaTeX for text processing, DOT for graphs drawing, SQL for databases and so on.

The reminder of this paper is structured in five sections. First, in Section 2, the characteristics of these languages are addressed. The main differences to GPLs are discussed in Section 3. The characteristics of these languages are intrinsically related to the advantages and disadvantages that they bring to their usage and development. These pros and cons are discussed in Section 4. Approaches and methodologies to the development of DSLs are discussed in Section 5. Finally, in Section 6, the paper is concluded.

2 DSLs Characteristics

GPLs are perfectly established in the software development life-cycle. Their characteristics are so widely spread among the software engineers community that are regarded as a natural thing, therefore, literature badly addresses this matter. However, with DSLs does not happens the same. There is the necessity of enhancing their characteristics in order to defend their usage instead of GPLs. In part, this happens because of the research and studies made on DSLs during the last ten years [7, 8, 5]; what reveals the

importance that these languages are achieving in software engineering. Because of that, the characteristics of DSLs are worthwhile to know and understand. The following paragraphs present those characteristics and justifies them.

Each DSL has unique characteristics, however the major part is common. In this context, it is possible to create a shelf for these languages, and evaluate and explain their common characteristics. In [9], the authors used the Cognitive Dimensions of Notations Framework (CDF) [10, 11] to accomplish a speculation-based evaluation of DSLs. In this section, the intention is to do a similar analysis. However in this case, the aim is to enhance, describe and point reasons for the characteristics of DSLs, regardless of the CDF.

Normally, DSLs are *small*. As these languages are tailored to deal with the problems of a specific domain, their designers can grab only the essential features and concepts of that domain, and manage them to create a small and restricted notation. That small notation allows the specification of solutions to solve the problems, instead of programming them, unlike what happens when dealing with the major part of GPLs. So, DSLs are more *declarative* or *descriptive* than imperative languages [4].

Moreover, they are *abstract* [12] and *expressive* [8]. When analyzing and designing a DSL, engineers should be aware of the domain where the language will be applied. The semantics of the domain should be implicit in the language notation [13]. The latter means that abstraction should be brought to the notation of the language; that is, the low level notions of how something is done should be encapsulated by a high level notation, expressing, for each sentence or statement, the precise purposes of their existence and usage. Indubitably, this allows their users to easily create mappings between the syntax of the language and the objects of the problem domain.

The concentration on the definition of a notation that would only express concepts of a single application domain, brings the possibility of sharpening edges on the language, and make it more and more *efficient* on various directions. One of these directions is the efficiency on being read and learned by the domain experts [6]. Domain experts are persons with great knowledge on a given domain, but, normally with any or little expertise on programming. Thus, given the abstractness and the expressiveness of DSLs, they can easily read programs, and learn the languages in order to specify the programs with efficiency, or in another words, with little time spent. Another facet of that efficiency can be observed on the tools that give support to the language. Their processors, for instance, can be improved to offer better results, as the domain is restricted and the knowledge is centralized.

3 Dichotomies on DSLs

DSLs are dichotomous. Their definition and characteristics divide the languages into two parts that can be regarded as a good and a less-good part. When analyzing these dichotomies it is inevitable the comparison with GPLs. For this discussion, the following pairs of characteristics are taken into account: i) *Abstractness vs Concreteness*; ii) *Low-Level vs High-Level* and iii) *Expressiveness vs Computational Power*.

3.1 Abstractness *versus* Concreteness

Abstractness and concreteness are subjective terms, because depend on the point of view, and on the objects that are being related. However, as stated before, programming is a human-oriented task in these days; so the perspective is always on the human side, and not on the computer's. Nonetheless, the computer is the object to relate with, in order to perform an analysis. So, what should be answered when is asked whether DSLs are abstract or concrete? A language is more abstract the more it hides the operational semantics [14] from the final user. One of the main efforts during the construction of a DSL, is to encapsulate the semantical knowledge in the language notation, in order to have it implicitly, instead of explicitly, as happens with GPLs. For this reason DSLs are more abstract than GPLs, which, by exclusion of parts, are more concrete.

In fact, as *Deursen* and *Klint* observed [15], the knowledge about the domain is concentrated in the language notation, and the knowledge about the implementation (the operational semantics of the language) is delegated to the compilers, processors or other related tools that give support to the language usage.

There are gains and losses on this matter. But surely the gains overcome the losses. Section 4, embodies a discussion about the advantages and disadvantages of using DSLs; there, the gains and losses of the abstract characteristic of DSLs, are addressed.

3.2 Low-Level *versus* High-Level

These dichotomous words are very similar to concreteness and abstractness, respectively. As a matter of fact, they depend on them. A low level language is a language that takes its user into a thinking level that is known to be unusual for a human beings. That is, the handling of constructors and abstractions of a low level language imply the presence of specialized knowledge beyond the empirical knowledge on an application domain. On the other hand, high level languages, remove from their notations explicit implementation aspects. Users are able, then, to handle the language constructors at a more rational level.

DSLs are high level languages, when comparing with GPLs. The abstractness of DSLs enable the rise of this level, as the semantics of the language are implicit in the constructors and abstractions that the notation of the language presents. This way, the user's concerns are focused on issues associated with the problem and not with the solution. The gains on following this philosophy are great; using high level languages (DSLs) that allow the focus on the problem and not in the solution, can be profitable at earlier stages of the software life-cycle [16], like the requirements analysis and management [17].

3.3 Expressiveness *versus* Computational Power

A language is said to be expressive when its notation helps the user on creating mappings between the program and problem domain concepts, without resorting to documentation of the software pieces, whether it is internal (comments, annotations and so forth) or external (user manuals, implementation reports and so forth). Such characteristic is observable when the language is easy to learn, and their programs are easy to write, read and understand.

On the other hand, the computational power of a language is related with the possibility of specifying multiple and different computations relying on the same vocabulary and structures, offered by the language. Also it is manifested when the notation allows the definition of similar computations, using different approaches.

DSLs are usually, more expressive than GPLs; on the other hand, GPLs offer a greater computational power. The major difference between these types of languages is precisely this dichotomy. However, it was not always true. Languages like Fortran or Cobol, can be regarded, nowadays, as GPLs, because of their computational power, but they were first constructed to fulfill requirements on the mathematics and the business areas, respectively [4]. That is, they were tailored for a single application domain.

The expressive power of DSLs make them very objective languages, in the extent that the user would know what happens when a statement of a program is interpreted or executed. The vocabulary of these languages store knowledge of the application domain, and mask behavioral aspects of this same domain. As they are used to cope with the aspects of a single domain, they do not need extra computational power to extrapolate for other domains. The language constructors must encapsulate precisely the behavior required for the concepts of the domain, with which they are associated.

As final words about this dichotomy, *Ladd* and *Ramming* [18] state that DSLs should not be designed to describe computations, but to express useful facts from which one or more computations can be derived. This express precisely what was said about DSLs through out the present section.

4 Advantages and Disadvantages

All the characteristics of DSLs, listed and explained before, imply a group of advantages, and obviously, some of them also a set of disadvantages.

In this section, an impartial and literature-based overview on this matter is given. In order to proceed, it is important to settle down two perspectives on this discussion. On the one hand, there is the *language usage* perspective, which is related with the usability of DSLs to produce, maintain, evolve or comprehend programs or specifications. On the other hand, there is the *language development* perspective, which concerns the implementation of processors (compilers or interpreters, editors, debuggers, animators, etc.) for the new DSLs.

4.1 On DSLs Usage Perspective

The most claimed advantage of using DSLs is the possibility of integrating domain experts in later stages of the software development life-cycle [15, 19]. Normally, domain-experts are very required in the analysis phase, and their functions end right there. But a few times, a new overview on the conceptual aspects concerning the software in production, is needed. Since the usage of GPLs require good programming skills, the domain-experts, who are not proficient on that area, little work can do on this matter. However, using DSLs they can steer the flow on the programming tasks and they can even give a hand on specifying the programs.

In this context, DSLs are also appropriated to diminish the distances that exist between the conception and implementation phases of the software development process. Also, this leads to the creation of new software development methodologies, meeting the requirements that the domain imposes [20].

Concerning the *software development process*, it is possible to sub-divide the present perspective into two parts: (i) the initial phases on the development process and (ii) the maintenance phase.

Concerning the former sub-perspective, (i), DSLs ease and increase the speed on the construction of software pieces [15]. *Kieburtz et al.* [21] also defined a set of advantages, that, in some extent, empower the existence of the last addressed advantage: they claim that DSLs enhance the flexibility, productivity and usability. The justification for these advantages is trivial by regarding the characteristics of DSLs presented before. All they are due, mainly, to the expressiveness and abstraction of these languages, which are, indeed, the main characteristics. Also, the usability of these languages can be justified by the fact of being small and easy to read and write.

DSLs also make easier the phase of testing in the software development process. In this context, non-traditional methods like [22] can be followed in order to test these programs.

Regarding the latter sub-perspective, (ii), there are many advantages. The main one is that using DSLs the software maintenance is simplified [15]. These languages enhance the comprehensibility of programs and specifications [7, 5]; which is not any novelty, because the easiness of maintaining a piece of software implies the easiness of comprehending the program specifications. Another important aspect of these languages is that, in many cases, they provide self-documentation, what avoids the search for documentation resources that may be unavailable. Together, these three aspects diminish the costs of engineering and reengineering, and increase reliability and repairability on the software constructed with DSLs [23].

DSLs are claimed to be a good approach for software reuse [24] — another advantage from the usage of DSLs. In this context, not only the pieces of software are reused, but also the knowledge embodied in the language.

Until now, only advantages were pointed. However DSLs have also some bad things associated, concerning their usage. Those presented below derive from the fact that DSLs are multi-characteristic languages. It is known that, several and distinct characteristics can difficult the adherence of users, because not all of them are used to cope with such variety of languages, and some programmers claim that do not need to learn many DSLs when already know one GPL. The point is, if there is not adherence on a DSL, it would not have any success, and its evolution or work done upon them, like tool support construction, would be tasks with no future; on the other hand, the adherence implies teaching costs, which can be very high [15]. Although the latter has been pointed as a disadvantage, it is believed that is easy to learn a DSL without effort.

Moreover, this variety reduces the easiness of interoperability with other well established languages [19]. This is obviously a disadvantage, because in real projects, several domains can be addressed and, thus, several DSLs can be used; and even the combination of DSLs with GPLs is a reality. So, without interoperability between them, DSLs can fall into disuse.

4.2 On DSLs Development Perspective

The development of DSLs allow optimization and validation at domain level [25]. The optimization of a language concerns with details of implementation at machine level, e.g. managing the memory allocation. As was stated before, these kind of implementations are done at language development time, and not at programming time, providing abstractions that encapsulate these details. So, at programming time, the user is able to build specifications and optimize them at domain level, because the optimization at machine level, is already done. For instance, in C, memory allocation is an indispensable task, but it takes the user to cope with details that are, most of the times, beyond their capabilities.

But this raises the major part of the disadvantages of developing DSLs. Firstly, in order to develop a language, the engineer must be expert on both the domain of application and compilers engineering [15]. Most of the times, the language engineers are not proficient on the problem domain, so creating a language is not a single-man task; domain experts are needed to steer the DSL requirements [26], and language engineers are needed to concretize the requirements into an abstraction capable of coping with the domain concepts.

This fact empowers the consumption of time and money on the several phases of the language development (design, implementation and maintenance) [4]. In fact, developing a language requires knowledge about programming in GPLs, on most part of the cases. The issues related with the maintenance of GPLPs are widely known. So, maintaing DSLs and their associated tools, like compilers, processors or interpreters, can be a very complicated and difficult task. Not for so little times, this leads to low number of tool supporting; and when there are tools, they may not follow up the evolutionary trends of the language [19].

Missing tool supporting for monitoring, debugging and other necessary tasks for DSL users is, indubitably, a great disadvantage. Without tools, the availability of DSL is limited [4], what leads to low number of adherents on such language; and the disadvantages of the last argument were already addressed in the previous section.

The disadvantages of the present perspective, can be attenuated regarding the methodologies used for the development of DSLs. Section 5 gives a little survey on some development methodologies that were successfully used to implement DSLs, over the times.

5 Developing DSLs

Language engineering is an old discipline, perfectly established as a branch of the software engineering. The development of DSLs is just a small part on that branch. Unlike the development of GPLs, which is, normally, based on compiler techniques [27], the development of DSLs follows several methods and techniques, including also the techniques used for creating GPLs [8].

Each development methodology is well supported by tools. Not that all the tools were tailored to cope with the development of DSLs but they are successfully and easily

adapted to cope with it. The DSLs that outcome from these different methodologies, are given a classification name. But regardless of that classification, they keep the same characteristics listed before and follow the same CDs.

In general, five main steps are pointed to be essential on DSL development: decision, analysis, design, implementation and deployment [8, 4]. The decision phase concerns with the analysis of pros and cons about developing or not a new language. The analysis is the phase where domain experts gather knowledge related to the domain, relying on domain analysis methodologies [28–30]. The design step is concerned with the choice and adoption of patterns and conceptual implementation decisions. The implementation phase, is related to the concrete construction of the DSL, using the patterns and implementation approaches adopted in the design phase. The deployment (or distribution) step is when the language is ready to be used, and is made available for general usage.

During these phases, patterns can be used to ease the process of language development. *Spinellis* [31], reinforced later by *Mernik et al.* [8], defined a set of several design patterns. These patterns are basis for implementation approaches. In [7], three main shelves were identified to contain such approaches. In the sections below are presented the two main containers and the approaches contained in each one.

5.1 Complete Language Design Approaches

The approaches contained in this shelf involve the creation of a language from the zero. This requires the definition of the language syntax (commonly achieved by using Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF) notations), the specification of the semantics, and the translation into target code. *Compilation* and *Interpretation* are the main approaches within this container.

Compilation — It is the most traditional approach to implement a language, no matter the language is a DSL or a GPL. The language constructs are analyzed and synthesized into a target code, that can be machine code or a common GPL language code, that is to be executed by a machine.

Interpretation — The difference to the compilation approach is that the language constructs are recognized but not executed at machine level; instead they are interpreted.

These implementation approaches are supported by compiler-compilers (tools for compiler generation) that are widely available for many years. The Lex and YACC [32, 33] system, used for the development of any kind of language, is one of the most popular. More recent and similar tools (in the extent that are not only focused on the development of DSLs) are available: JavaCC[3], SableCC [34], based on translation grammars, and LISA [35], AntLR [36], JastAdd [37], based on Attribute Grammars (AG) [38], are some examples. On the other hand, there are tools specialized on the construction of compilers and interpreters for DSLs. Draco [39], ASF+SDF [40], Kephera [41], Kodiyak [23], InfoWiz [42], are some examples of such tools.

The languages created with these approaches are said to be *external languages* [43], because they are independent languages; that is, the DSL is completely designed and implemented from the scratch, not relying on any pre-existent language.

[3] Home page: https://javacc.dev.java.net/

5.2 Languages Extension Approaches

In this container are placed the approaches for DSLsimplementation using GPLs, or components of these languages, as a start point. In this context, two approaches are considered to be the most used:*Embedding* and *Extensible compiler/interpreter* approaches.

Embedding — In these approaches, the developer does not need to have compilers expertise, because all the work of semantics verification and target code transformation is delegated to the compiler of the base language. Nevertheless, the constructs of the new DSL must be designed, using the base language syntax. The construction of an Application Programming Interface (API) for a given application domain, is considered a concretization of this approach, and is the several times used.

The advantages of embedding languages are considerable. From the fact that it needs no compiler knowledge, to the fact that the compiler of the base-langauge is completely reused; from the point that the development of the language needs only knowledge on the domain and on the base language, to the point that, for usage purpose, it is not need to know the underlying language.

The languages implemented with this approach are called *internal* or *embedded languages* [12]. Almost any GPL can be used as base-language, but some have more appropriated characteristics and are frequently used: Ruby [44], Python [45], Haskell [12, 46], Java [47], C++ [48] and Boo [43] are some examples of utilization.

A specialization of the embedded languages was introduced by *Fowler* and *Evans* [49], and is called *Fluent Interfaces*. These languages still being classified as embedded, but their construction upon a GPL follows a methodology enabling a more flexibility in the syntax. So writing operations is no more than chaining function calls, what allows a more fluent specification.

Extensible compiler/interpreter — This approach is very similar to the compilation approach presented in Section 5.1. The main difference is that an existent compiler of a GPL is extended with domain-specific aspects in order to add domain-specific constructs to the underlying language, rather than creating the compiler from the scratch. This way, the task of creating the language is easier than using the other approach.

Summing up this discussion about the implementation of DSLs, *Kosar et al.* [5], concluded that the extension of languages by the embedding approach is the most efficient and effective approach to be used when developing DSLs. They also claim that a great alternative to this approach is to use the compilation approach, because of the possibility of handling minor aspects and having more control over the language implementation.

Another shelf identified is using Commercial Off-The-Shelf (COTS) products. The idea on this approach is to specify the language structural aspects in terms of these tools [7]. As this approach is not so consensual, no more details about it are given.

6 Conclusion

The usage of Domain-specific Languages (DSLs) is not a novelty. Since the beginning of programming languages, DSLs have been created to focus the programmer on the particularities of a concrete problem domain, and not on the programming details. Thus,

a great number of DSLs have been tailored for a considerable amount of domains. *Mernik et al.* [8] provide several examples of existing DSLs and identify, also, their application domains.

The fact of DSLs being designed for a specific domain confers them a reasonable number of characteristics that make them different from GPLs and more competitive. However, to get efficient implementations, the development of their processor (compiler, interpreters, etc.) is not an easy task and requires systematic approaches based on traditional compiler construction technology. Nevertheless, these difficulties can be soften regarding the variety of implementation approaches that have been successfully used.

Although not yet corroborated by the literature, the community believes that DSLs are much more user-friendly, and their programs are easier to comprehend, when comparing with GPLs. Experiments and work on program comprehension for DSLs and their usability have been taken off in the context of DSLpc[4]. There are already some raw results on this matter, described in some papers recently submitted to conferences on the area.

The objective of this paper was to summarize a review of DSLs literature. Particular attention was paid to the theoretical perspective associated with these languages, rather than to the practical one, as have been done for the last ten years. Basilar characteristics, advantages and disadvantages were pointed out. In addition, approaches and methodologies for their development were surveyed, for a complete overview of the topic under discussion.

References

1. Watt, D.A.: Programming language concepts and paradigms. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990)
2. Hutchings, B.L., Nelson, B.E.: Using general-purpose programming languages for FPGA design. In: DAC '00: Proceedings of the 37th conference on Design automation, New York, NY, USA, ACM (2000) 561–566
3. Visser, E.: WebDSL: A case study in domain-specific language engineering. In Lammel, R., Saraiva, J., Visser, J., eds.: Generative and Transformational Techniques in Software Engineering (GTTSE 2007). Lecture Notes in Computer Science, Springer (2008)
4. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices **35** (2000) 26–36
5. Kosar, T., López, P.E.M., Barrientos, P.A., Mernik, M.: A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology. **50**(5) (April 2008) 390–405
6. Consel, C., Latry, F., Réveillère, L., Cointe, P.: A generative programming approach to developing dsl compilers. In Gluck, R., Lowry, M., eds.: Fourth International Conference on Generative Programming and Component Engineering (GPCE). Volume 3676 of Lecture Notes in Computer Science., Tallinn, Estonia, Springer-Verlag (sep 2005) 29–46
7. Wile, D.S.: Supporting the dsl spectrum. Journal of Computing and Information Technology **9**(4) (2001) 263–287

[4] DSLpc is a collaborative project between Portugal and Slovenia, and stands for *Program Comprehension for Domain-specific Languages*. More details can be found in http://epl.di.uminho.pt/~gepl/DSL/

8. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys. **37**(4) (December 2005) 316–344
9. Pereira, M.J.V., Mernik, M., da Cruz, D., Henriques, P.R.: Program comprehension for domain-specific languages. ComSIS – Computer Science an Information Systems Journal, Special Issue on Compilers, Related Technologies and Applications **5**(2) (Dec 2008) 1–17
10. Blackwell, A., Britton, C., Cox, A., Green, T.R.G., Gurr, C., Kadoda, G., Kutar, M., Loomes, M., Nehaniv, C., Petre, M., Roast, C., Roe, C., Wong, A., Young, R.: Cognitive dimensions of notations: Design tools for cognitive technology. In: Cognitive Technology: Instruments of Mind. Springer-Verlag (2001) 325–341
11. Green, T.R.G., Blandford, A.E., Church, L., Roast, C.R., Clarke, S.: Cognitive dimensions: achievements, new directions, and open questions. Journal of Visual Languages & Computing **17**(4) (August 2006) 328–365
12. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys **28**(4) (June 1996) 196–202
13. Hudak, P.: Modular domain specific languages and tools. In: ICSR '98: Proceedings of the 5th International Conference on Software Reuse, Washington, DC, USA, IEEE Computer Society (1998)
14. Nielson, H.R., Nielson, F.: Semantics with applications: a formal introduction. John Wiley & Sons, Inc., New York, NY, USA (1992)
15. van Deursen, A., Klint, P.: Little languages: little maintenance? Technical report, University of Amsterdam, Amsterdam, The Netherlands (1997)
16. Jackson, M.: Problem frames and software engineering. Information and Software Technology **47**(14) (November 2005) 903–912
17. Aurum, A., Wohlin, C.: Engineering and Managing Software Requirements. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
18. Ladd, D.A., Ramming, C.J.: Two application languages in software production. In: VHLLS'94: USENIX 1994 Very High Level Languages Symposium Proceedings, Berkeley, CA, USA, USENIX Association (1994) 10
19. Gray, J., Fisher, K., Consel, C., Karsai, G., Mernik, M., Tolvanen, J.P.: Dsls: the good, the bad, and the ugly. In: OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, New York, NY, USA, ACM (2008) 791–794
20. Anlauff, M., Chaussee, R., Kutter, P.W., Pierantonio, A.: Domain specific languages in software engineering (1998)
21. Kieburtz, R.B., Mckinney, L., Bell, J.M., Hook, J., Kotov, A., Lewis, J., Oliva, D.P., Sheard, T., Smith, I., Walton, L.: A software engineering experiment in software component generation. In: ICSE '96: Proceedings of the 18th international conference on Software engineering, Washington, DC, USA, IEEE Computer Society (1996) 542–552
22. Sirer, E.G., Bershad, B.N.: Using production grammars in software testing. In: PLAN '99: Proceedings of the 2nd conference on Domain-specific languages, New York, NY, USA, ACM (1999) 1–13
23. Herndon, R.M., Berzins, V.A.: The realizable benefits of a language prototyping language. IEEE Transactions on Software Engineering **14**(6) (1988) 803–809
24. Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2) (June 1992) 131–183
25. Menon, V., Pingali, K.: A case for source-level transformations in matlab. In: PLAN '99: Proceedings of the 2nd conference on Domain-specific languages, New York, NY, USA, ACM (1999) 53–65
26. Kolovos, D.S., Paige, R.F., Kelly, T., Polack, F.A.C.: Requirements for domain-specific languages. In: Proc. 1st ECOOP Workshop on Domain-Specific Program Development (DSPD 2006), Nantes, France (July 2006)

27. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley (August 2006)
28. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute (November 1990)
29. Simos, M.A.: Organization domain modeling (ODM): formalizing the core domain modeling life cycle. SIGSOFT Softw. Eng. Notes **20**(SI) (1995) 196–205
30. Frakes, W., Diaz, R.P., Fox, C.: DARE: Domain analysis and reuse environment. Ann. Softw. Eng. **5** (1998) 125–141
31. Spinellis, D.: Notable design patterns for domain-specific languages. Journal of Systems and Software **56**(1) (February 2001) 91–99
32. Lesk, M.E., Schmidt, E.: Lex - a lexical analyzer generator. UNIX: research system **2** (1990) 375–387
33. Johnson, S.C.: Yacc: Yet another compiler compiler. In: UNIX Programmer's Manual. Volume 2. Holt, Rinehart, and Winston, New York, NY, USA (1979) 353–387
34. Gagnon, E.M., Hendren, L.J.: SableCC, an object-oriented compiler framework. In: Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings. (1998) 140–154
35. Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V.: Lisa: An interactive environment for programming language development. In: Compiler Construction. Springer Berlin / Heidelberg (2002) 1–4
36. Parr, T., Quong, R.W.: AntLR: A predicated-LL(K) parser generator. Software Practice and Experience **25**(7) (July 1995) 789–810
37. Ekman, T., Hedin, G.: The jastadd extensible java compiler. SIGPLAN Not. **42**(10) (2007) 1–18
38. Knuth, D.E.: Semantics of context-free languages. Theory of Computing Systems **2**(2) (June 1968) 127–145
39. Neighbors, J.: The draco approach to constructing software from reusable components. Readings in artificial intelligence and software engineering (1986) 525–535
40. van Deursen, A., Heering, J., Klint, P.: Language Prototyping: An Algebraic Specification Approach: Vol. V. World Scientific Publishing Co., Inc. (1996)
41. Faith, R.E., Nyland, L.S., Prins, J.F.: Khepera: a system for rapid implementation of domain specific languages. In: DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997, Berkeley, CA, USA, USENIX Association (1997) 19–31
42. Nakatani, L.H., Jones, M.A.: Jargons and infocentrism. In: In First ACM SIGPLAN Workshop on Domain-Specific Languages. (1997) 59–74
43. Rahien, A.: Building Domain-Specific Languages in Boo. Manning (February 2008) (Note: Unedited Draft).
44. Cunningham, C.H.: A little language for surveys: Constructing an internal DSL in Ruby. In: Proceedings of the ACM SouthEast Conference, Auburn, Alabama (March 2008)
45. Paul, R.: Designing and implementing a domain-specific language. Linux J. **2005**(135) (2005) 7+
46. Peterson, J.: A language for mathematical visualization. In: Proceedings of FPDE'02: Functional and Declarative Languages in Education. (2002)
47. Kabanov, J., Raudjärv, R.: Embedded typesafe domain specific languages for java. In: PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java, New York, NY, USA, ACM (2008) 189–197
48. Prud'homme, C.: A domain specific embedded language in c++ for automatic differentiation, projection, integration and variational formulations. Sci. Program. **14**(2) (2006) 81–110
49. Fowler, M., Evans, E.: Fluent interfaces (December 2005) Web Reference: `http://www.martinfowler.com/bliki/FluentInterface.html`; Visited on May 2009.