2IMP20 - Domain Specific Language Design
Assignment 3
Individual Report

Dmitrii Orlov, 1068971

11 July 2022

Introduction

For the final assignment for this course we have decided to design a domain specific language (DSL) called Agriculture Irrigation Language, AIL for short. This language is intended to fit in the domain of Precision Agriculture [], by focusing on its monitoring aspect. Precision Agriculture allows to use technical means to manage the farmer's assets and observe their conditions in near real-time []. We build on top this idea, and define the language syntax usable specifically for this purposes and provide the implementation of a parser for this language implemented with Flex/Bison []. The parser not only allows to recognize the files written in AIL, but also translate this DSL into an equivalent executable Python code. Further in this report, I take an individual perspective on this assignment. In the first section, I reflect on the process, implementation results, and the key learnings of this project. In the second section, I analyse 3 papers related to the process of this assignment.

1 Language designer perspective

Creating a new language is not a trivial, but is a very enjoyable process. During this course, we have studied two main perspectives on domain specific language design and implementation: grammarware and modelware [19]. The former takes a text-first approach, with the design and implementation of the language syntax taking the leading role. The latter, takes the models as a central concept and approaches the design of the language through domain modeling. During the course we have experienced both the grammarware and modelware approaches in practice by using two different language workbenches (LWBs) Rascal [18] and Eclipse Modeling Framework (EMF) [17] respectively. Design and implementation of LWBs is an active topic in research and industrial communities and has even been formally specified as the "Language Workbench Challenge" (LWC) in 2010 at the Code Generation conference [9]. However, this area in particular is a subdomain of a larger field defining a paradigm of Language Oriented Programming (LOP) [12]. Arguably, at the very heart of LOP, lays the "compiler of compilers" Lex/Yacc [15]. Lex/Yacc is a compiler generator tool, created in 1970s, being effectively one of the oldest tools designed for language development [10, 11]. Flex/Bison is a more modern GNU tool suite upwards compatible with Lex/Yacc [1, 3].

Curious to discover what is the experience with working with the classical approach, that led to the eventual emergence of so many newer techniques and given a free choice of the language development tools, we have decided to use Flex/Bison. Primarily, this was motivated by a pure interest in discovering a tool that withstood the test of time and has gained a wide popularity and

continues to be used to this day, that also allowed us to disregard its limitations in comparison to newer techniques and even in comparison to the reportedly easier to use parser generator tool ANTLR [15]. Furthermore, the secondary motivation was a desire to purposefully unwrap the built-in features of the many LWBs [9] to understand the underlying techniques even better and have a first hand experience with a tool that misses them, even if to simply appreciate their presence in tools like Rascal and EMF.

1.1 Flex and Bison

Flex/Bison tool suite, at its core, comes as two CLI tools: `flex` and `bison` respectively [11]. These tools are agnostic to the development environments, such as Integrated Development Environments (IDEs), and are intended to solely be used within a command-line environment. For getting familiar and finding our way through Flex/Bison, we relied on the official documentation, and the available open source community discussions. Its minimalist set of dependencies allows a large flexibility in how the language-oriented project will be developed and features it will include, providing, on the one hand, a large feeling of freedom, but also control over the project, but on the other hand, introducing all the challenges of developing any other project, with addition of challenges caused by the Flex/Bison familiarity. Further in this section, dive deeper into our development setup for working on this project, and analysing its benefits and challenges in more detail.

1.1.1 Development utilities

To facilitate the development of Flex/Bison projects in the environment of your choice, there exist a large number of third party libraries and extensions. These tools include syntax highlights for Flex and Bison files in popular IDEs, linting, error checking, code auto-completion, and warning displays directly in the source code. For our convenience, we have decided to use Visual Studio Code (VS Code) as our IDE, and to enable the convenient workspace for Flex and Bison, we have used an extension called Yash [6], providing all the features listed above.

Furthermore, since Flex/Bison files may include arbitrary C code [11], VS Code IDE provides a robust set of features simplifying the development tasks for this language. To work on our project, we have relied on the Microsoft's official C/C++ extension pack [2]. It includes features like syntax highlighting, debugging and execution of C code, static analysis and so on.

Generally, a Flex/Bison project includes two individual sources working together to produce a final parser executable: lexer (also known as tokenizer) implemented with Flex, and the parser implemented with Bison. In order to produce the final executable both sources need to be compiled to a common C source, which thereafter can be made into an executable [11]. Lexer's responsibility is to recognize the language constructs in the provided file, returning individual tokens on every match. The parser, using these tokens, defines the effective syntax of the DSL using Backus-Naur Form (BNF) grammars [13]. The Bison internal syntax, allows to define *actions* that will be executed upon a successful match of a token in this grammar. This code can be implemented in C, using all features offered by C. Often this is used for semantic interpretation of the parsed input [11]. In order to compile the full executable parser that will recognize the input and perform the assigned actions, we have configured a compilation step using GCC compiler for C [4].

To further simplify repeated actions, we have created a GNU Make Makefile [5], that contained the different development related targets.

1.1.2 Benefits

As the key positive takeaway from working with Flex/Bison, we observe the added value of freedom of choosing your own development environment. The modern IDEs provide a wide range of supportive tools helping the developers make their work easier, more efficient, and less error prone [7]. This indeed has been observed during the development, allowing to use the wide range of features related not solely to Flex/Bison tooling, but also to editing supporting documentation, automation of tasks within the development flow, editing additional files and mixing different languages for diverse source code in the same environment (Flex sources, Bison sources, C sources, but also Python, and Make).

Additional advantage of Flex/Bison, in our opinion, is its minimal dependency requirement and the flexibility of producing arbitrary code in a widely used general purpose language, C. This allows to use all the best practices and techniques of developing other software in C. Such techniques naturally include code reuse, relying on standard and third-party libraries to simplify the development work, debugging, and even using Foreign Function Interface to extend and reuse functionality of other languages [20].

1.1.3 Challenges

However, on the flip side, there are also disadvantages and challenges in using Flex/Bison. First of all, in our experience, the ease of development of Flex/Bison was capped by the ease of development of C programs. Certainly, an experienced C programmer shall not find this a difficulty, but for novices, this comes as a larger friction point. The difficulty of the "host" language, C, in this case, makes the development more complicated and error prone, causing more time needed to debug simple features and focus on technicalities of the implementation of the DSL semantic operations, as opposed to focusing primarily on the language design itself. In this sense, our experience concurs with the observations of Ortin et al., suggesting that Flex/Bison is not the most optimal tool choice for novice language developers [15].

Secondly, reflecting back on the previous experience with more feature rich LWBs, like Rascal and EMF, the availability of certain language-related features out of the box, allows to save time on implementation of very generic aspects. Namely, implementation of syntax highlighting, to our knowledge, appeared difficult to provide for Flex/Bison-based DSLs, with the closest possibility in sight being to create a separate embedded DSL within Flex sources to extract the highlighting information. Although this would provide an possibility to produce arbitrary many flavours of syntax highlighting, it would also almost double the amount of required work and eventual maintenance.

Thirdly, to our knowledge, we could not find an easy way to benefit from reuse of the generic syntax configurations for our language with Flex/Bison, which could, otherwise save time in defining very simple building blocks. To give one example where this could come most convenient, is the semantic interpretation of standard data structures (such as *Lists*, *Tuples*, *Key-Value Maps*, etc.) and defining attribute access expressions.

1.2 Implementation Results

As the result of this project, we did manage to implement a functioning parser for the AIL language. Originally, we have identified a number of intended features for the language in three categories: *core*, *useful* and *bonus*.

1.2.1 Core features

The intended core features included the following items:

- Full AIL grammar + syntax parsing
- AIL semantic correctness
- AIL well-formedness checks
- AIL to Python translation
- Python executable mimics the intended AIL behavior

Out of which, all items have been completed on the basic level, with some details being ignored due to the time constraints. Most noticeable, is the ignoring of the provided attribute values and action body specification during the generation of the Python code.

1.2.2 Useful features

- Consistent and user friendly parse error reporting
- Auto-formatting of AIL files (pretty printing)

Out of these features only the top feature has been implemented.

1.2.3 Bonus features

- Syntax-highlighting file generation
- Extended static analysis and checking for AIL
- Extended syntactic sugar for AIL

Out of these features none have been implemented.

1.3 Learnings & Discussion

As our key learning, having used Flex/Bison for this project allowed us to get a better understanding of the complexity of the techniques underlying the DSL development. We have experienced both benefits and drawbacks of using Flex/Bison, that will allow us to make a more informed decision on choosing the LWBs for future projects focusing on using the right tool for the right job. And lastly, we have experienced that creating a functioning DSL language is possible using regular development techniques and practices in an iterative fashion gradually building up the features surrounding the language.

2 Related work reflection

In this section we provide a brief analysis of 3 DSL-related papers.

2.1 "Towards supporting SPL engineering in low-code platforms using a DSL approach", Bragança et al., 2021

In their paper, Bragança et al. address an important question of reuse of language components during the DSL implementation [8]. In this paper the extend on the result of their earlier work on the SCOLAR framework. This framework provides a possibility to organize DSL components into "product lines" and reuse these pieces to create new languages through code generation. However, such approach is limited in that it can not support loosely coupled reuse. Their current contribution is in extending the framework to overcome this limitation by defining a new set of concepts in the metamodel of SCOLAR.

Loose coupling, in this case refers to using only a subset of aspects of different programming languages - for example identifiers, but not the full underlying models. This can be used in cases where well-formedness constraint validation is sufficient, without considering deeper structures. Their proposal allows to define language aggregations exposing specific features of the languages. These aggregations are further arranged into language "families", which through an interface model, allow to define the "exposed" and the "required" language features. They further propose nine requirements for the language aggregations ensuring the safety and soundness of the aggregation. Further, they describe a reuse workflow through "DSL Component Compositiong Workflows".

Their extension to the metamodel, provides a more granular control over the reuse. Namely, the language developers now can specify to which extend the component structure should be borrowed and where. They describe the new metamodel and validate its conformity to the nine requirements identified on the original metamodel of SCOLAR.

Through their case study, they demonstrate a reuse of components to define a new "Automation" language using the MontiCore LWB. Thereby the demonstrate both the feasibility and correctness of their implementation. The main benefit of this extended SCOLAR metamodel is two-fold, on the one hand it facilitates reuse of components without knowing their internal structure to a great detail through the granular aggregation access, and, on the other hand, by providing a more informative way to declare and reason about the reuse enabling the language experts in deciding which specific properties are needed or not.

They finalize their paper with identifying a number of limitations of their approach and encouraging further work on (1) providing more utilities facilitating implementation and reasoning about the reuse, and (2) providing support of SCOLAR in a broader range of client LWBs.

2.2 "Domain-Specific Languages for IoT: Challenges and Opportunities", Salman et al., 2021

In their work, Salman et al. [16] conduct a study on the usage of the DSLs in IoT domain and suggest a taxonomy of DSLs separating on 5 criteria. They contribute a more thorough systematisation of the existing DSLs by providing a granular perspective to classify the languages. They observe that majority of the DSLs are used to manage custom IoT applications in the real world scenarios. They observe a dominating lack of visual interfaces for the DSLs and major focus of the DSL design on IoT-experts rather than on the *field* experts. They argue that this leads to more fragmented DSL landscape and negatively impacts the reliability and trust factor in the IoT environments. They encourage applying more user-driver design techniques and prioritizing the reliability and user-friendliness of the DSLs.

2.3 "Towards Ontology-based Domain Specific Language for Internet of Things", Negm et al., 2020

The work of Negm et al. also relates to the use of DSL's in IoT domain, similar to the previous paper [14]. They also contribute to the discourse on involving more domain expertise in the language design process and call for making the DSL user friendlier and more trustworthy. They propose the technique of basing the DSL development on introduction of domain-specific "ontologies" describing all the involved entities of the domain. They propose a language called OntIoT, allowing to use the ontology-driven techniques for implementation of new DSLs. They suggest, that this approach, eliminates the need of creating generic components shared by many languages in the ontology-driven domains. Their works is most directly inspirational for our purposes due to its match with out target domain and purposes.

References

[1] Bison - gnu project - free software foundation.

[2] C/c++ extension pack - visual studio marketplace.

[3] Flex: the fast lexical analyzer has moved.

[4] Gcc, the gnu compiler collection - gnu project.

[5] Make - gnu project - free software foundation.

[6] Yash - visual studio marketplace.

[7] Zakieh Alizadehsani, Enrique Goyenechea Gomez, Hadi Ghaemi, Sara Rodríguez González, Jaume Jordan, Alberto Fernández, and Belén Pérez-Lancho. Modern integrated development environment(Ides). In Juan M. Corchado and Saber Trabelsi, editors, *Sustainable Smart Cities and Territories*, Lecture Notes in Networks and Systems, pages 274–288, Cham, 2022. Springer International Publishing.

[8] Alexandre Bragança, Isabel Azevedo, Nuno Bettencourt, Carlos Morais, Diogo Teixeira, and David Caetano. Towards supporting SPL engineering in low-code platforms using a DSL approach. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 16–28, Chicago IL USA, October 2021. ACM.

[9] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, volume 8225, pages 197–217. Springer International Publishing, Cham, 2013.

[10] Michael E Lesk and Eric Schmidt. *Lex: A lexical analyzer generator*. Bell Laboratories Murray Hill, NJ, 1975.

[11] John R. Levine. *Flex & bison*. O'Reilly, Sebastopol, CA, 1st ed edition, 2009. OCLC: ocn321016664.

[12] David H. Lorenz and Boaz Rosenan. Code reuse with language oriented programming. In Klaus Schmid, editor, *Top Productivity through Software Reuse*, volume 6727, pages 167–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[13] Daniel D. McCracken and Edwin D. Reilly. *Backus-Naur Form (BNF)*, page 129–131. John Wiley and Sons Ltd., GBR, 2003.

[14] Eman Negm, Soha Makady, and Akram Salah. Towards ontology-based domain specific language for internet of things. In *Proceedings of the 2020 9th International Conference on Software and Information Engineering (ICSIE)*, ICSIE 2020, page 146–151, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Francisco Ortin, Jose Quiroga, Oscar Rodriguez-Prieto, and Miguel Garcia. An empirical evaluation of Lex/Yacc and ANTLR parser generation tools. *PLOS ONE*, 17(3):e0264326, March 2022.

[16] Aymen J Salman, Mohammed Al-Jawad, and Wisam Al Tameemi. Domain-specific languages for IoT: Challenges and opportunities. *IOP Conference Series: Materials Science and Engineering*, 1067(1):012133, feb 2021.

[17] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[18] Tijs van der Storm. The rascal language workbench. *CWI. Software Engineering [SEN]*, 13:14, 2011.

[19] Manuel Wimmer and Gerhard Kramler. Bridging grammarware and modelware. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, Lecture Notes in Computer Science, pages 159–168, Berlin, Heidelberg, 2006. Springer.

[20] Jeremy Yallop, David Sheets, and Anil Madhavapeddy. A modular foreign function interface. *Science of Computer Programming*, 164:82–97, October 2018.