A Language Specification Tool for Model-Based Parsing

Luis Quesada, Fernando Berzal, and Juan-Carlos Cubero

Department of Computer Science and Artificial Intelligence, CITIC,
University of Granada, Granada 18071, Spain
{lquesada,fberzal,jc.cubero}@decsai.ugr.es

Abstract. Typically, formal languages are described by providing a textual BNF-like notation specification, which is then manually annotated for syntax-directed translation. When the use of an explicit model is required, its implementation requires the development of the conversion steps between the model and the grammar, and between the parse tree and the model instance. Whenever the language specification is modified, the developer has to manually propagate changes throughout the entire language processor pipeline. These updates are time-consuming, tedious, and error-prone. Besides, in the case that different applications use the same language, the developer has to maintain several copies of the same language specification. In this paper, we introduce a model-based parser generator that decouples language specification from language processing, hence avoiding many of the problems caused by grammar-driven parsers and parser generators.

Keywords: Language specification, parser generator, Model-Driven Software Development (MDSD).

1 Introduction

Formal languages allow the expression of information in the form of symbol sequences [3]. A formal language consists of an alphabet, which describes the basic symbol or character set of the language, and a grammar, which describes how to form valid sentences in the language. In Computer Science, formal languages are used for the precise definition of the syntax of data formats and programming languages, among other things.

Most existing language specification techniques [2] require the developer to provide a textual specification of the language grammar. The proper specification of such a grammar is a nontrivial process that depends on the lexical and syntactic analysis techniques to be used, since each kind of technique requires the grammar to comply with different restrictions.

When the use of an explicit model is required, its implementation requires the development of the conversion steps between the model and the grammar, and between the parse tree and the model instance. Thus, in this case, the implementation of the language processor becomes harder.

Whenever the language specification is modified, the developer has to manually propagate changes throughout the entire language processor pipeline. These updates are time-consuming, tedious, and error-prone. This hampers the maintainability and evolution of the language [11].

Typically, different applications that use the same language are developed. For example, the compiler, different code generators, and the tools within the IDE, such as the editor or the debugger. The traditional language processor development procedure enforces the maintenance of several copies of the same language specification in sync.

In contrast, generating a model-based language specification is performed visually and does not require the development of any conversion steps. By following this approach, the model can be modified as needed without having to worry about the language processor, which will be automatically updated accordingly. Also, as the software code can be combined with the model in a clean fashion, there is no embedding or mixing with the language processor. Finally, as the model is not bound to a specific analysis technique, it is possible to evaluate the alternative or complementary techniques that fit a specific problem, without propagating the restrictions of the used analysis technique into the model.

Our approach to model-based language specification has direct applications in the following fields:

– The generation of language processors (compilers and interpreters) [1].
– The specification of domain-specific languages (DSLs), which are languages oriented to the domain of a particular problem, its representation, or the representation of a specific technique to solve it [7,8,17].
– The development of Model-Driven Software Development (MDSD) tools [21].
– Data integration, as part of the preprocessing process in data mining [22].
– Text mining applications [23,4], in order to extract high quality information from the analysis of huge text data bases.
– Natural language processing [9] in restricted lexical and syntactic domains.
– The corpus-based induction of models [12].

Although there are tools that generate language processors from graphical language specifications [19,6], to the best of our knowledge, no existing tool follows the approach we describe in this paper.

In this paper, we introduce ModelCC, a model-based tool for language specification. ModelCC acts as a parser generator that decouples language specification from language processing, hence avoiding many of the problems caused by grammar-driven parsers and parser generators.

2 Background

Formal grammars are used to specify the syntax of a language [1].

Context-free grammars are formal grammars in which the productions are of the form $N \rightarrow (\Sigma \cup N)^*$ [3]. These grammars generate context-free languages.

A context-free grammar is said to be ambiguous if there exists a string that can be generated in more than one way. A context-free language is inherently ambiguous if all context-free grammars generating it are ambiguous.

Typically, language processing tools divide the analysis into two separate phases; namely, scanning (or lexical analysis) and parsing (or syntax analysis).

A lexical analyzer, also called lexer or scanner, processes an input string conforming to a language specification and produces the tokens found within it. A syntactic analyzer, also called parser, processes an input data structure consisting of tokens and determines its grammatical structure with respect to the given language grammar, usually in the form of parse trees.

Traditional efficient parsers for restricted context-free grammars, as the LL [20], SLL, LR [14], SLR, LR(1), or LALR parsers [1], do not consider ambiguities in syntactic analysis, so they cannot be used to perform parsing in those cases. The efficiency of these parsers is $O(n)$, being n the token sequence length.

Existing parsers for unrestricted context-free grammar parsing, as the CYK parser [24,10] and the Earley parser [5], can consider syntactic ambiguities. The efficiency of these parsers is $O(n^3)$, being n the token sequence length.

Lex and *yacc* [15] are well-known lexer generator and parser generator, respectively. It is difficult to specify all the constructions of a language in BNF-like notation without causing conflicts that these tools do not support [1].

ANTLR [18] is a lexer and parser generator that allows the generation of tree parsers. Tree parsers are recognizers that process abstract syntax trees instead of symbol sequences. This tool generates an LL(*) parser, which does not accept ambiguous grammar specifications either.

YAJco [19] is a lexer and parser generator that accepts as input a set of Java classes with annotations that specify the prefixes, suffixes, operators, tokens, parentheses and optional elements. This tool generates a BNF specification for JavaCC [16], which is a lexer and parser generator that supports LL(k) grammars. Therefore, the developer still has to be careful so the grammar implicit in the Java class set complies with the LL(k) grammar restrictions.

3 Model-Based Language Specification

We introduce ModelCC, a model-based tool for language specification that generates a language processor from a model.

3.1 Abstract syntax versus concrete syntax

The abstract syntax of a language is just a representation of the structure of the different components of a language without the superfluous details related to its particular textual representation [13]. On the other hand, concrete syntax is a particularization of the abstract syntax that defines, with precision, a specific textual or graphical representation of a language. It should be noted that a single abstract syntax can be shared by several concrete syntaxes.

For example, the abstract syntax of the typical *if-then-optional else* sentence of an imperative programming language could be specified as a composition of a condition and one or two sentences. Two concrete syntaxes corresponding to specific textual representations of such a conditional sentence could be specified as: {"if", "(", expression, ")", sentence, and optionally "else" and another sentence}, and {"IF", expression, "THEN", sentence, optionally "ELSE" and another sentence, and "ENDIF"}.

When using ModelCC, the language designer has to focus on the language abstract syntax model instead of focusing on specifying the BNF-like notation that describes a concrete syntax.

The advantages of this approach have been widely studied [13]:

– Specifying the abstract syntax seems to be a better starting point than specifying a concrete syntax.
– The language designer is able to modify the abstract syntax model and generate a working IDE on the run.
– It is not necessary for the developer to have advanced knowledge on parser generators to develop a language interpreter. In particular, the developer will not need to face the restrictions these parser generators usually impose.
– Priorities and associativity restrictions between elements that can cause ambiguities can be effortlessly established and modified.

3.2 Metamodel-based approach versus traditional approach

A diagram summarizing the traditional language specification procedure is shown in Figure 1. It illustrates the requirements of giving a BNF-like language specification and converting it into an attribute grammar. It also shows the lack of an explicit representation of the abstract syntax model, and the fact that the concrete syntax is the starting point of the process.

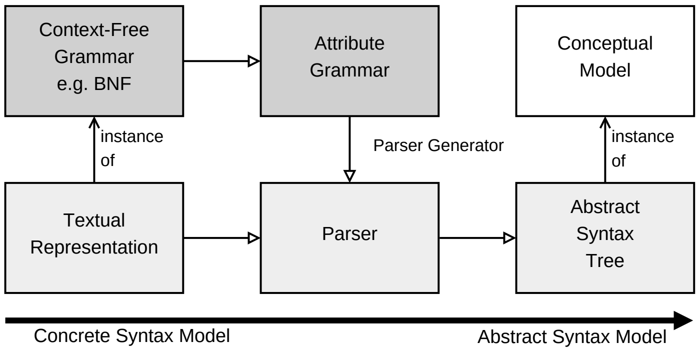

Fig. 1. Traditional language processing approach.

A diagram summarizing the model-based language specification approach used by ModelCC is shown in Figure 2. Developer workload is reduced as it just involves defining an abstract syntax model, which is annotated to automatically generate the grammar of the concrete syntax and its corresponding parser.

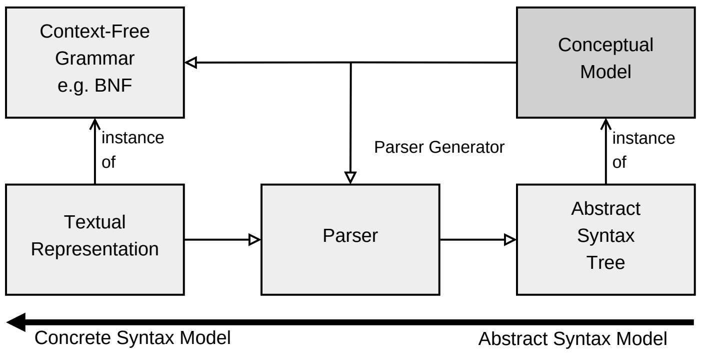

Fig. 2. Our approach to model-based language specification and processing.

3.3 Model specification

ModelCC provides the developer several mechanisms that can be used to create a model. Two of them are typical in model specification: inheritance and composition. The rest are annotations that complement the model elements by specifying patterns, delimiters, cardinality, and evaluation order. A summary of the annotations supported by ModelCC is shown in Figure 3

Constraints on...	Annotation	Usage
Patterns	@Pattern	Pattern matcher for a specific element.
	@Value	Field where the matched text should be stored.
Delimiters	@Prefix	Element prefix(es).
	@Suffix	Element suffix(es).
	@Separator	Element enumeration separator(s).
Cardinality	@Optional	Composition optionality.
	@Minimum	Minimum element multiplicity.
	@Maximum	Maximum element multiplicity.
Evaluation order	@Associativity	Element associativity (e.g. left-to-right).
	@Composition	Eager or lazy constructions.
	@Priority	Element precedence level/relationships.

Fig. 3. Summary of the annotations supported by ModelCC.

4 Benefits of Model-Based Language Specification

As a simple example of the expression power of ModelCC, we have specified a simple calculator-like language that supports the following constructions:

- Unary operators: +, and -.
- Binary operators: +, -, *, and /.
- The binary operators * and / share the higher precedence.
- The binary operators + and - share the lower precedence.
- Parenthesis can be used to enforce precedence.
- Integer and real number support, althought results are always real numbers.

The model-based specification of this language is shown in Figure 4.

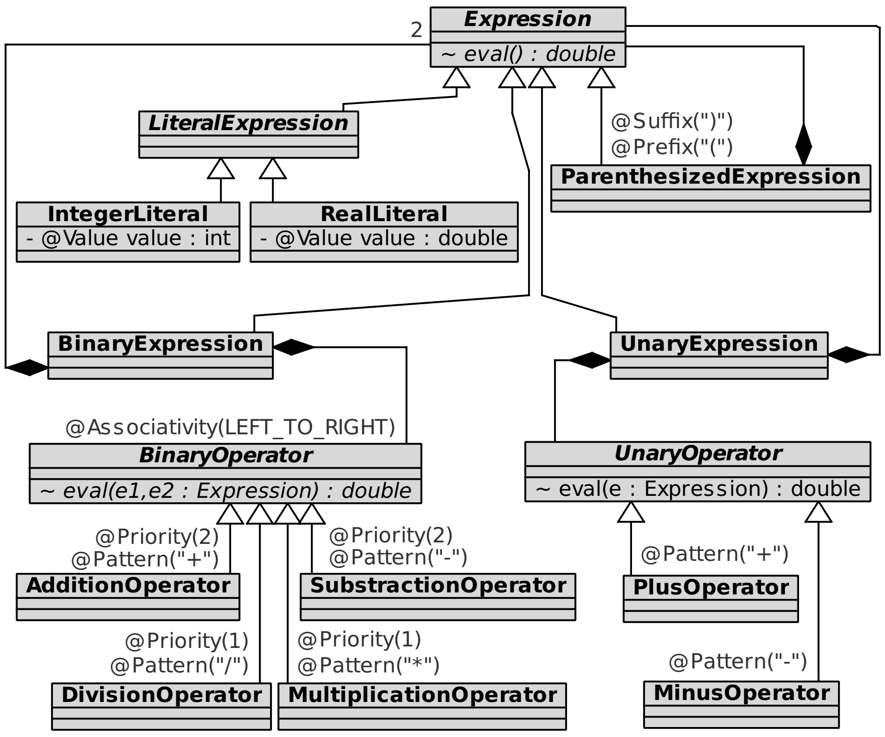

Fig. 4. ModelCC specification of a simple calculator language.

Besides the model-based approach, the main functional advantages of using ModelCC over other existing tools such as *lex/yacc*, *YAJCo*, or *ANTLR* are the following:

- Apart from regular expressions, ModelCC allows the usage of pattern matching classes, which can be coded for specific purposes. For example, a dictionary-

based matcher, in contrast to a regular expression-based matcher, could be used for detecting verbal forms in ModelCC.
– ModelCC supports multiple composition constructions. There is no need to bring the BNF-like notation recursion of enumeration specifications to the model.
– ModelCC offers a generic associativity and priority mechanism instead of a specific and limited operator specification mechanism. It supports creating operator-alike constructions as complex as needed. For example, it allows the usage of non terminal symbols as operators and defining n-ary operators.
– ModelCC provides mechanisms that allow the developer to solve most language ambiguities. For example, expression nesting ambiguities can be solved by using associativities and priorities, and *if-then-optional else* sentence nesting ambiguities can be solved by using composition restrictions.

5 Conclusions and Future Work

We have introduced ModelCC, a model-based tool for language specification that automatically generates a parser from a model representing the abstract syntax of the language.

ModelCC automates several steps of the language processor implementation process and it improves the maintainability of languages.

Moreover, ModelCC allows the reuse of a language specification among different applications, eliminating the duplication required by conventional tools and improving the modularity of a language processing tool set, since it allows the use of object-oriented design techniques to cleanly separate language specification from language processing.

It should be noted that the ModelCC approach is not bound to any particular lexical or syntactic analysis technique. ModelCC models do not need to comply with the constraints imposed by particular parsing algorithms.

A fully-functional "proof of concept" implementation of ModelCC is soon to be released at the www.modelcc.org website.

In the future, ModelCC will incorporate a wider variety of parsing techniques so that it will be able to automatically determine the most efficient parsing technique that does not incur in ambiguities for processing a particular language.

ModelCC will also be extended in order to support multiple concrete syntaxes (for a single abstract syntax).

Finally, we also plan to study the use of ModelCC in different application domains, including model induction, natural language processing, text mining applications, data integration, and information extraction.

Acknowledgements: Work partially supported by research project TIN2009-08296.

References

1. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 2nd edition, 2006.

2. A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing & Volume II: Compiling.* Prentice Hall, Englewood Cliffs, N.J., 1972.
3. N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–123, 1956.
4. V. Crescenzi and G. Mecca. Automatic information extraction from large websites. *Journal of the ACM*, 51:731–779, 2004.
5. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 26:57–61, 1983.
6. H. Ehrig and G. Taentzer. Graphical representation and graph transformation. *ACM Computing Surveys*, 31:9, 1999.
7. M. Fowler. *Domain-Specific Languages.* Addison-Wesley Signature Series (Fowler), 2010.
8. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28:196, 1996.
9. D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition.* Prentice Hall, 2nd edition, 2009.
10. T. Kasami and K. Torii. A syntax-analysis procedure for unambiguous context-free grammars. *Journal of the ACM*, 16:423–431, 1969.
11. L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10)*, pages 918–932, 2010.
12. D. Klein and C. D. Manning. Corpus-based induction of syntactic structure: Models of dependency and constituency. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics (ACL '04)*, pages 479–486, 2004.
13. A. Kleppe. Towards the generation of a text-based ide from a language metamodel. volume 4530 of *Lecture Notes in Computer Science*, pages 114–129, 2007.
14. D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.
15. J. R. Levine, T. Mason, and D. Brown. *lex&yacc.* O'Reilly, 2nd edition, 1992.
16. C. McManis. Looking for lex and yacc for java? you don't know jack, 1996. JavaWorld, www.javaworld.com/javaworld/jw-12-1996/jw-12-jack.html.
17. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37:316–344, 2005.
18. T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1995.
19. J. Porubän, M. Forgáč, and M. Sabo. Annotation based parser generator. In *Proceedings of the International Multiconference on Computer Science and Information Technology, IEEE Computer Society Press*, volume 4, pages 705–712, 2009.
20. D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top-down grammars. *Information and Control*, 17:226–256, 1970.
21. D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39:25–31, 2006.
22. P.-N. Tan and V. Kumar. *Introduction to Data Mining.* Addison Wesley, 2006.
23. J. Turmo, A. Ageno, and N. Cataà. Adaptive information extraction. *ACM Computing Surveys*, 38:4, 2006.
24. D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10:189–208, 1967.