Architectural Design Patterns for Language Parsers

Gábor Kövesdán, Márk Asztalos and László Lengyel

Budapest University of Technology and Economics
Department of Automation and Applied Informatics
Magyar tudósok krt. 2, H-1117 Budapest, Hungary
{gabor.kovesdan, mark.asztalos, laszlo.lengyel}@aut.bme.hu

Abstract: Processing the textual scripts of computer languages is an important field in software development, which has been growing in popularity, recently. It is applied both for general-purpose programming languages and for domain-specific languages. There is a wide range of typical algorithms and patterns that are used to syntactically parse formal languages, each having specific characteristics and implying different software architectures. If we develop parsers at a higher abstraction level, it simplifies the problem domain and facilitates developing more robust software quicker, but there are always some tradeoffs to consider. The main guideline of this paper is abstraction: how to increase it in different patterns, how it helps parser development and what kind of tradeoffs are implied. The presented architectural design patterns are organized in a pattern catalog ordered by their abstraction level. This catalog is intended to assist developers in the industry in designing efficient parser software.

Keywords: modeling; model-processing; formal languages; parsers; design patterns; architectural patterns

1 Introduction

The complexity of systems is growing at a rapid pace, which promotes the use of modeling and formal methods in today's software environment. When modeling is used for a well-defined field and has a limited expressive nature, we usually speak about domain-specific modeling [1] [2]. This limited expressive potential reduces the problem domain and thus simplifies and facilitates software engineering. We can express models in two common ways: with graphical or textual representations. The latter has led to processing of textual scripts and formal language theory being used more widely than in the past [3]. Writing parsers for computer languages was formerly a very specific knowledge that was only necessary to design and implement compilers but was not a commonly required

skill. This situation has changed recently as software developers are more frequently facing the problem of processing textual representations of models. Although processing of domain-specific modeling languages is usually based on the theoretical background of compilers for general-purpose programming languages, their limited expressive power sometimes can lead to a simpler syntax, which allows other, often simpler, parsing approaches. This work presents processing methods of both types.

Despite the increasing use of domain-specific languages, we still lack catalogs that collect best practices for this field. Since the introduction of design patterns in [4], developers have a catalog of generally applicable object-oriented best practices and a universal terminology to refer to commonly used software recipes. This book had a great influence on the software development area, but it only contained a wide or general focus. Catalogs of more specialized patterns are still very much in demand, in order to collect more ideas and to create a common vocabulary among developers. In the field of domain-specific languages, [1] provides a pattern catalog, covering several different aspects of domain-specific languages. This is a rich source of information but it has a more general view than this paper and does not provide such a systematic organization of architectural patterns used in parsing as intended herein. Apart from this, [5] provides some practical uses of general object-oriented design patterns in recursive descent parsers and [6] describes how a parser generator uses object-oriented design patterns. These are just specific uses of general design patterns and these papers do not include more specialized patterns specific to parsers.

The goal of this paper is to fill the gap and provide a pattern catalog of design patterns that can be used to design parser software. We focus on architectural-level design patterns, which determine the software components and their interaction. Although some of the patterns described herein result in an identical runtime component structure, we judged that it was practical to describe them, because the development techniques and workflow that they suggest are very different. This also leads to deviating consequences, so we extended the notion of architectural design pattern to possibly include some aspects of development workflow. In fact, [1] also distinguishes some patterns that result in an identical runtime structure, though it does not explicitly merit this generalization. When creating the pattern catalog, we examined the interests of all actors within the industry that are concerned with parsing software: end-users, domain specialists, developers and tool developers. They have different focuses and goals, which were taken into account in the description of patterns, so that all actors can benefit from using this catalog.

The rest of this paper is organized as follows. In Section 2, the theoretical background is explained, which is necessary to understand the concepts outlined in the paper. In Section 3, the actual pattern catalog is presented. We provide an introductory section that summarizes its deliverables to different actors of the industry and then patterns are listed according to their level of abstraction. We

have intentionally avoided the terms maturity and evolution and consider it important to explicitly highlight that a higher level of abstraction does not necessarily guarantee a specific pattern is consistently more effective or practical. Higher abstraction usually means that less development work is necessary, but sometimes it may also include less flexibility.

2 Background

In the following, we summarize the principal concepts of parsing. This theoretical background is explained in more detail in [7] and [8].

First, we discuss the processing of textual scripts, we must divide this process into two major steps: parsing and execution. Parsing [1] is described here as a step in which the input is read and mapped to an internal representation, the semantic model, which is more meaningful than any intermediate abstract representation. Execution refers to the actual processing of the model, which can mean running some executable steps or a transformation executed on the model. Before describing the actual patterns, we briefly discuss the possibility of parsing and executing languages with increasing levels of abstraction. In software engineering, we identify some kind of systematic organization or commonly applicable steps, factor them out as reusable units and parameterize them with the parameters that change from one use case to the other. Abstraction is increased in this way. Later on, we discuss how this applies to language processors.

The least structured parser that we can imagine is a monolithic component, which reads an input and executes it immediately. One commonly used intuitive parsing solution that follows a more systematic approach is a delimiter-directed translation [1]. With this approach, the language is divided into chunks, called tokens, by identifying well-recognizable separator characters. Then the actual type of each token can be inferred from its position. For some simple languages, this is a quick and easy solution that can be developed. However, for a more complicated language, the notion of grammar [7] must be introduced. Grammar is a formalism that describes how the language constructs can be used together to create well-formed words[1]. The grammar uses two kinds of symbols: a terminal symbol is a symbol that can occur in well-formed words, while a non-terminal symbol is only used for substitutions, in an attempt to derive well-formed words from the grammar. There is a special non-terminal, the start symbol, from which the derivation starts. A grammar is composed of production rules, which have two sides and express a possible substitution while deriving a well-formed word. The

[1] A well-formed word is a statement expressed in the language that is considered valid and meaningful.

left side is the series of symbols that are substituted from the right side. If we can find a derivation from the start symbol that uses the production rules and successfully derives the word we are validating, then the particular word is considered well-formed in the language.

To simplify the process, production rules are usually phrased at the level of the tokens and not at the single characters. We construct a lexical analyzer (also called lexer or scanner) that creates a series of tokens from the input character stream. The syntactical analyzer then uses this token stream and a parsing algorithm to verify whether the word is well-formed in the language. While using the production rules to derive the input word, it also stores the parse tree or a simplified version of that, which is usually denoted as an abstract syntax tree. When the distinction between them is irrelevant, they are usually referred to as syntax trees. This phase is sometimes ambiguously referred to as parsing, identical to the entire process. With the help of the notion of grammar, we now have a more systematic method to build a parser: create a grammar, write a lexer that produces a token stream from the input and write the syntactical analyzer that uses the production rules of the grammar to decide whether the input belongs to the language. At the same time, it builds a syntax tree and if the derivation succeeds, this syntax tree can be used for the semantic analysis of the input. Figure 1 summarizes how a script is processed.

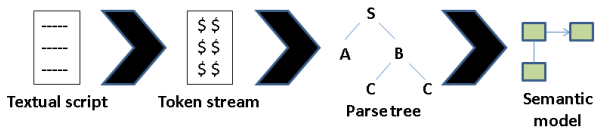


Figure 1
The processing steps of parsing a language

Since the parsing algorithms are generally applicable to a wide set of grammars (context-free grammars with some further limitations), the so called parser generators or compiler compilers (sometimes also referred to as compiler generators) have been developed to automatically generate parsers from a grammar in a well-specified notation. Using these tools, we can create a ready-to-use parser that provides us with a syntax tree that can be used to further process the language. This highly simplifies the development effort for a parser. Nevertheless, the syntax tree is still not the semantic model (although in some cases it may be) so a third pass, a tree parser, is required. By further increasing the abstraction, it is possible to avoid manually developing the tree parser as well. Upon agreeing on the metamodel of the semantic model, it is possible to construct tools that directly generate a parser, that can map the input language to the semantic model. This entails that the grammar language has to be extended to include additional information about the methods for mapping language constructs, to model elements and their properties. This way, we can generate all components needed to parse the language and build the required semantic model.

3 Pattern Catalog

The different approaches for the parsing process that have been presented in Section 2, can be described as architectural design patterns. This section provides a catalog of them using a format similar to [4]. Similarly, because of their higher-level nature, these patterns do not have such a specific intent as general design patterns and therefore, the intent section has been renamed to *Description*. Furthermore, the sections on alternative names and related patterns have also been removed. In *Description*, the main idea of the discussed architectural structure is explained. *Motivation* describes the main factors for choosing the pattern. *Applicability* enumerates the criteria required for the pattern to be relevant when applied. In *Structure*, the composition and collaboration of the parser components are explained and represented with a summarizing figure. The *Consequences* section highlights the deliverables and warns about potential disadvantages of the pattern. In particular, the following aspects are examined: (1) the intermediate data structures used in the pattern and their memory consumption; (2) the clarity of the resulting component structure and the ease of modifications and extensions; (3) the level of expressiveness in the language that the pattern allows; (4) the reusability; (5) the extent of required compiler-specific knowledge; (6) the amount of code to be hand-written and the development time required and (7) the effort required to build and integrate the components in the system under construction. *Implementation* provides some guidelines or best practices to consider when employing the architecture. Last but not least are the *Known Uses* section, it presents an example and provides references, so that the application and implementation of the pattern can be grasped and later studied more efficiently.

Table 1 can be consulted for a short summary of the actors in the field of computer languages and their motivation in using this pattern catalog.

The figures included in the pattern descriptions use a loosely formalized, but intuitive graphical notation that explains the component structure and the data flow among components. We have not found a standard notation for this purpose that was intuitive enough. Our notation uses the following conventions: Program components are represented with rounded rectangles that include a “meshed gear or cogwheel” icon. Components supplied by third-party vendors have a double border. Generated components have a dashed border. Components that have to be hand-written have a single border. Input and output artifacts and intermediate data structures are represented by an arbitrary icon and are explained in the caption. Arrows show the information flow between components. Arrows with the gear icon denote that a component generates another.

In the following, these patterns will be discussed:

1. Ad-hoc Parser
2. Delimiter-Directed Parser

3. Multiple Pass Parser
4. Common Carrier Syntax
5. Parser Generator
6. Semantic Mapper Generator

Table 1
Motivations of actors in using the pattern catalog

Role	Focus	Expectations	Motivation
User	Software product	The specified requirements should be satisfied by the final software product.	Choosing the right architectural design can facilitate satisfying the requirements.
Domain Specialist	Input and output models	The expressiveness of the textual domain-specific language used in the system is high enough to properly model domain-specific artifacts and the syntax is intuitive and easy to use. Output models may have to meet similar criteria.	Architectural patterns influence the expressiveness of the language and the way they can be processed.
Developer	Code and component structure	The software can be developed in an easy and quick way. The code is easy to understand, to extend and to maintain.	The architectural pattern influences complexity and maintainability.
Tool Developer	Reusable tools	Providing market-ready tools that can be reused in real-life software products and can facilitate rapid software development.	Several pattern explained here are based on increasing abstraction by reuse. A good tool should leverage the advantages of these patterns while mitigating the weaknesses.

3.1 Ad-hoc Parser

Description The parser is a single monolithic unit. There is no explicit notion of grammar nor are there consecutive subtasks.

Motivation The complexity of the language does not require a higher-level pattern and it is desirable to avoid the development overhead of that.

This solution is often applied, if the performance of the system is so critical that higher-level patterns cannot be used because of their additional runtime overhead.

Applicability This kind of parser is only applicable with simple languages. If the language is such that it can be executed on-the-fly, we can keep memory usage low. On-the-fly processing means that the fragments of the script can be processed immediately, as they are read, and there is no need to store significant information about the context.

Structure The parser itself is a single unit that reads the input and processes it. It does not contain any common structural units. Figure 2 depicts an ad-hoc parser.

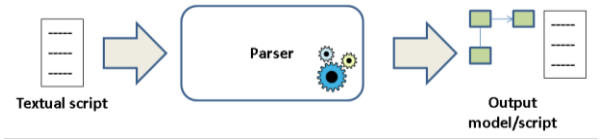


Figure 2
An ad-hoc parser

Consequences Because the parser is a single unit, there is no need for communication among components or building intermediate data structures. This reduces memory consumption and execution time.

Because of the lack of inner structures, the parser code may be difficult to understand, maintain and extend. If the language is likely to change in the future, the maintenance cost of monolithic parsers drastically increases.

The lack of inner structures and a systematic engineering process also implies that only low-complexity systems can be developed in this way. This greatly limits the expressiveness of the language.

The parser does not have reusable components but the low complexity leads to less time spent on its development.

Building an ad-hoc parser does not require specific knowledge in language theory or compiler engineering.

Implementation Because of the ad-hoc nature of the solution there is no general implementation structure. The specifics of the processed language must be checked in order to realize a practical implementation.

Known Uses In [9] Brian Kernighan demonstrates a simple implementation of the grep command-line utility. The grep utility expects a regular expression and looks for matching lines in the given input. The implementation explained by Brian Kernighan uses a limited subset of standard POSIX regular expressions but solves the matching problem on the fly with a minimal memory footprint. The code is apparently easy to understand because of the low complexity of the problem but once more advanced regular expression features are required, extension becomes a difficult task.

3.2 Delimiter-Directed Translation

Description The parser searches for separator symbols that separate the tokens of the input language script. The type of each token is then inferred by its position. The parser may build a semantic model in the memory, but it may also be possible to execute the language script on the fly.

Motivation The simple structure of the language does not warrant building a full-fledged parser that creates a substantial syntax tree in the memory. This can save memory and reduce execution time. The development overhead of syntax-directed parsers is also avoided.

Applicability The language can easily be split into tokens by looking for separator symbols and tokens can easily be interpreted by their position.

Structure The parser remains a major component, but it may also build a semantic model and pass it to an execution component. Figure 3 shows the structure of a delimiter-directed parser.

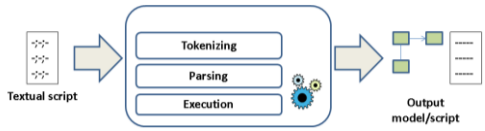


Figure 3
A delimiter-directed parser

Consequences The parser is fast, simple and has a minimal memory footprint because the lack of separate components avoids communication overhead and creating intermediate data structures.

Its code is somewhat easier to maintain, extend and understand when compared to ad-hoc parser since there is already a stream of clearly distinguished tokens, which are usually reflected in the code as array indexes. Although most languages use several different delimiters, if the syntax is more complex, it is not really easy to capture it with the delimiter-driven approach and the code also becomes difficult to read and modify. This limits the expressiveness of the language under design.

The parser does not have significantly reusable components but the low complexity leads to less time spent on coding. If structured well, the tokenizer may be reused.

Building a delimiter-directed parser does not require specific knowledge in language theory or compiler engineering.

Implementation Delimiter-directed parsers are frequently coded with a loop that reads statements one by one. This usually means reading the input line by line. Then, within a loop, the lines are split into tokens that are practically stored in an array or a list structure. Most programming languages also provide tokenizer functions. This may be followed either by populating a semantic model or a direct function call that processes the tokenized information.

Known Uses Comma-separated values (CSV) is a common file format that enumerates properties of entities. Each entity is described by its own line and the property values follow a specific order and are separated by commas or semicolons. Such a file can easily be processed by a delimiter-directed parser.

3.3 Multiple Pass Parser (Syntax-Directed Translation)

Description The notion of grammar is used to systematically create the parser that attempts to derive the input word from the production rules. The parser is usually divided into different logical components that prepare and perform this derivation and then map the resulting syntax tree to the semantic model. The number of the components may vary from parser to parser.

The development of the parser is driven by the grammar of the language but it is possible that the resulting code does not explicitly express the grammar.

Motivation The language is too complex, and parsing it with an ad-hoc parser or a delimiter-directed parser would be challenging.

Facilities for better maintenance and extensibility are required.

Applicability The pattern is always applicable; nevertheless the specific parsing algorithms have different limitations regarding the type of grammars they can handle. In general, a large subset of context-free grammars can be handled by choosing the appropriate algorithm. Parsing algorithms have different attributes with regard to the generality of the grammar, performance and memory consumption.

Structure The parser is composed of several components, and the following three are the most common: (1) the lexical analyzer, (2) the syntactic analyzer and the (3) tree parser.

A lexical analyzer provides a token stream for a syntactic analyzer, which in turn, builds a syntax tree. The syntax tree is quite a low-level construct that reflects the structure of the language itself but usually not the semantic meaning of the model. A third component may be used to map the syntax tree to the semantic model. Most often, a semantic model is provided although in some cases, the syntax tree may function as a semantic model as well. In Figure 4, we can see the workflow between the components of a multiple-pass parser.

Consequences Implementing the parser requires knowledge in compiler development.

Based on the chosen algorithm, the code may not explicitly reflect the grammar.

The parser can handle a wide subset of context-free grammars, the concrete limitations of which depend on the particular algorithm.

Based on the theory of formal languages, the parser can be algorithmically efficient. However, there are intermediate data structures used between the consecutive passes, which means that building them requires computational time and they increase the memory footprint.

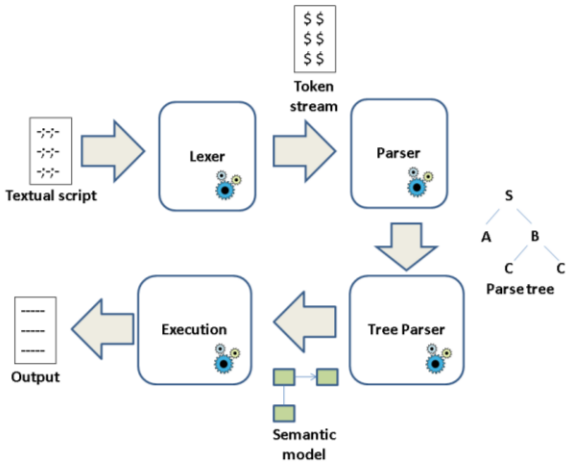


Figure 4
A multiple pass parser

Because of the systematic construction facilitated by the notion of grammar, the code is easier to maintain, to extend and to understand. The code may not always explicitly reflect the grammar, so the cost increases with future changes.

The separation of concerns achieved by the decomposition of the parser requires a lot of coding efforts and these components are only moderately reusable.

Implementation The lexical tokens are usually described by a regular language, which has limited expressiveness but this still increases abstraction and thus facilitates the implementation of the syntactic analyzer. The syntactic analyzer works with context-free languages and therefore can capture more aspects of the language than the lexical analyzer. In automata theory, context-free languages are recognized by push-down automata. It is not surprising that many implementations therefore use a stack data structure. A large set of algorithms uses a table-driven approach that requires a parsing table, which determines the action to take by examining the topmost stack symbol and the current input token. Other algorithms use recursive calls to capture a LIFO (Last In First Out) behavior, such as the recursive descent parser or the recursive ascent parser [10] [11].

Known Uses Most hand-written parsers of complex languages use multiple passes to simplify parsing and parser generators usually generate code using this abstraction.

3.4 Common Carrier Syntax (Syntax-Directed Translation)

Description The language syntax is composed of a limited set of language constructs, which makes it possible to reuse an existing parser. The available language constructs have a general nature so that a wide range of domain-specific languages can be expressed with it. An optional grammar description can further limit the syntax. In this context, such optional grammar will be referred to as vocabulary or schema. The parser then may have two inputs: the script to be processed, and its schema description.

Motivation The reuse of the fixed set of language constructs also makes the toolchain heavily reusable.

Using more uniform syntaxes among different languages facilitates standardization of the syntax and lowers the learning curve for new users.

The limited set of language constructs can be defined in such a way that facilitates efficient parsing.

Applicability The syntax of the language under design fits into the syntax that available common carrier syntaxes provide. This is usually the case when the language is read more often than modified. In this case, conciseness, in which these syntaxes usually fail, is not such an important aspect.

Identical or similar carrier syntax is used for other domain-specific languages in the environment where the language under construction is being developed. End users of the language are already familiar with the carrier syntax, which lowers the learning curve of the new language and the existing toolchain can be reused when it comes to processing scripts of a new language.

Structure From an architectural viewpoint, the inner design of the parser is irrelevant. It can be considered as a black box that has one or two inputs: the optional schema and the language script to parse. Depending on the intent of the tool, it may provide rich functionality of processing and transformations or simply the minimum required to transform the script into a syntax tree or any other language-independent data structure that will reflect how the elements of the language script relate to each other. Figure 5 shows how common carrier syntaxes are used to process textual scripts.

Consequences Implementing a language in common carrier syntax is easy; it consists of specifying how the previously defined syntax will be used to express model elements. This can be done with an informal description or with a schema language that places further limitations on the syntax, depending on the functionality provided by the underlying common carrier syntax.

Common carrier syntaxes may sometimes be too verbose, since the fixed elements of the syntax add some syntactic noise. This also has a negative effect on the readability by humans. The extent of this depends greatly upon the actual common

carrier syntax. This is often unwanted and a more concise and intuitive syntax is preferred despite the advantages of using a common carrier syntax.

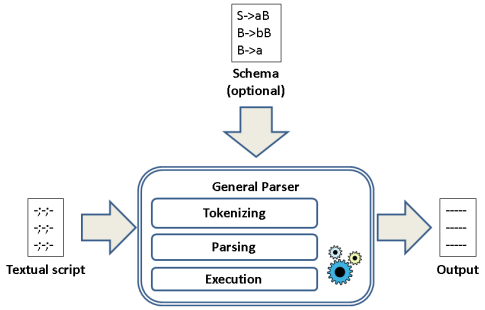


Figure 5
A parser of a common carrier syntax

Languages implemented in common carrier syntaxes are very easy to extend by modifying the (implicit or explicit) schema.

Using a common carrier syntax allows heavily reusing the parser and other tools used for language processing. Depending on our choice of the common carrier syntax, there may be numerous free and commercial implementations of parsing and processing tools. Because of the generality and constant development, these tools tend to be quite mature and efficient. Nevertheless, different processing libraries may have different APIs and different tools may be invoked with different options. This can make it practical to use a build management tool to integrate components. If there are good third-party tools available, coding may be required only for integration.

The standardization lowers the learning curve for each actor involved in the development and use of the software product under construction.

Implementation One well-known standard for common carrier syntaxes is the eXtensible Markup Language (XML) [12]. It has several schema languages available to define the vocabulary of our domain-specific languages, although not all of them are supported in each parser. It has a high number of related standards that assist in the transformation and processing of XML documents: XSLT [13], XPath [14], XQuery [15], and so on. XML is frequently criticized because of its verbose syntax, which is not the most convenient to write or read manually without specific tooling. On the other hand, its broad support and rich functionality make it a stable foundation for domain-specific languages.

Two possible alternatives to XML are YAML [16] and JSON [17]. Their syntax is more concise and they are also standardized. However, they lack the accompanying transformation and processing standards, and that highly reduces their functionality. They also lack standard means of specifying a schema, although there are third-party solutions to define constraints.

Known Uses MathML [18] is an application of XML to create a language that can describe mathematical formulae. The language itself is defined using the XML Document Type Definition (DTD) schema language. Browsers and rendering software that support MathML leverage existing XML parser libraries. Although the syntax of MathML is quite verbose, the XML nature of it makes it possible to simply embed formulae into web pages or to process them with XML tools.

3.5 Parser Generator (Syntax-Directed Translation)

Description A parser generator software is used to generate a multiple pass parser from the grammar description. The grammar is specified within the own notation of the parser generator and this description is used to generate a parser using a widely applicable parsing algorithm. Production rules in the grammar can usually contain the so-called semantic action routines that are code fragments written in the target programming language. These code fragments are generated into the output code and are run when a particular language element is recognized. It makes possible for the programmer to build a syntax tree or to populate a semantic model.

Motivation The language is complex enough to warrant a syntax-directed parser; however, either the development team is not skilled in writing parsers or the parser does not need to be highly optimized or customized. Using a parser generator, the development time of the parser is usually low.

Applicability The language can be parsed with the available general parsing algorithms and the performance of such a solution will suffice. Also, there are available parser generators that suit the requirements or building such is affordable.

Structure The parser generator takes the grammar description, which may contain additional information about generation parameters, for example, target programming language in which the parser is implemented, the package name in case of Java code, and so on. The generator then outputs one or more software components that are used to parse scripts and build a syntax tree or a semantic model. In Figure 6, we can see a system that uses a parser generator.

Consequences The build process of the parser is more complex because of the code generation step.

The programmer must write the grammar that provides a higher-level view of the language and does not have to deal too much with the lower-level code. One exception is writing the semantic action routines if the default parse tree built by the generated parser needs to be customized. This lowers the learning curve, limits hand-written code and allows for the rapid development of parsers.

Semantic action routines are foreign code in the grammar description. Since they are factored out from their context, they cannot be properly validated. Besides, scattering also makes these code pieces difficult to understand. The overall effect of them may not be easily recognized, which can make the development process more error-prone and the debug process more difficult.

The generated code is usually a multiple-pass parser, which means that the memory footprint will be high because of the large data structures used in the parsing process.

The parser generator tool is highly reusable.

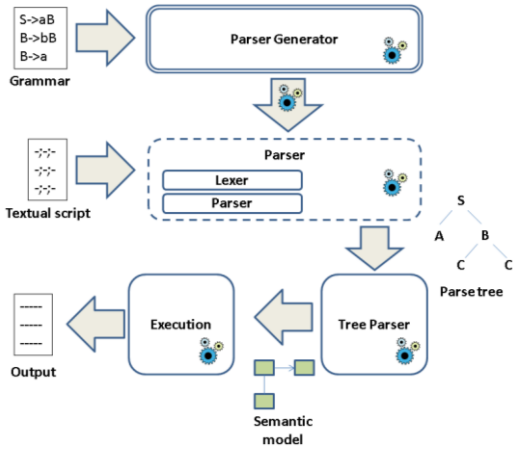


Figure 6

A parser system using a parser generator

Implementation There is a vast diversity among parser generators. They differ in the target language(s) they generate, in the grammar notation used to specify the syntax and in the particular parsing algorithm they use (which also implies different limitations on the syntax). They may generate a lexer or may also support the use of a hand-written lexer. Some parser generators generate table-driven code and fill in parsing tables according to the specified grammar while others may generate a recursive algorithm. Some parser generators may also support so-called syntactic predicates [19] which are additional notations intermixed with the grammar and can control certain parsing-time behavior that cannot be inferred from the production rules of the grammar.

Known Uses ANTLR [20] [21] [22] is one of the most popular parser generator tools, which is implemented in Java but can generate parsers in several target languages. It generates an LL(*) [23] parser, which is a top-down parser with arbitrary look-ahead length, then implements it with a recursive descent approach. It handles a wide subset of context-free grammars, although, because of the top-down nature, it cannot handle left-recursion.

3.6 Semantic Mapper Generator (Syntax-directed Translation)

Description Suppose that we already have an appropriate metamodel and we want to use a parser generator because of the advantages discussed earlier. Since the metamodel is fixed, it is possible to build such a parser generator that also generates a tree parser, not just a syntactical analyzer. The grammar has to be extended with additional mapping notation that specifies how language constructs are mapped to elements of the semantic model. This solution allows us to define this mapping at a higher level and to generate code instead of manually developing the tree parser.

Motivation Having a parser that directly outputs a semantic model increases abstraction and reduces hand-written code by leveraging generative programming [2].

Semantic action routines used in parser generators (see Section 3.5) lead to scattered code, which is difficult to read and the overall effect is not obvious; therefore this is an error-prone technique. Generating a parser that directly populates the semantic model overcomes this.

Applicability The metamodel of the semantic model is known, and there exists a tool that can map the language elements directly to the semantic model or, if this is not possible, the development of such a tool is an affordable option.

Structure The semantic mapper generator takes a grammar with semantic mapping information. It generates a semantic mapper that goes through the parsing process and builds an in-memory semantic model. This model is an instance of the used metamodel and also reflects the model information stored in the language script. The generated semantic mapper may be a monolithic unit or may be implemented as consecutive passes of parsing, like lexical analysis, syntactical analysis and tree parsing. The architecture that uses a semantic mapper generator is depicted in Figure 7.

Consequences The build process of the parser is more complex due to the code generation step.

Since the semantic model is built by the generated semantic mapper, there is no need for extra programming, such as using semantic action routines. This lowers the learning curve, eliminates hand-written code in the parsing phase and allows for rapid development of language parsers.

The generated high-level code may impede customizing finer-grained, inner behavior and thus may have an average performance or memory footprint.

The metamodel used by existing semantic mapper generators may be limiting and the alternative, developing a new semantic mapper generator for a custom metamodel may lead to (significantly) higher costs.

The generated code is usually a multiple-pass parser, resulting in a larger memory footprint for the data structures required for parsing.

The semantic mapper generator tool is highly reusable.

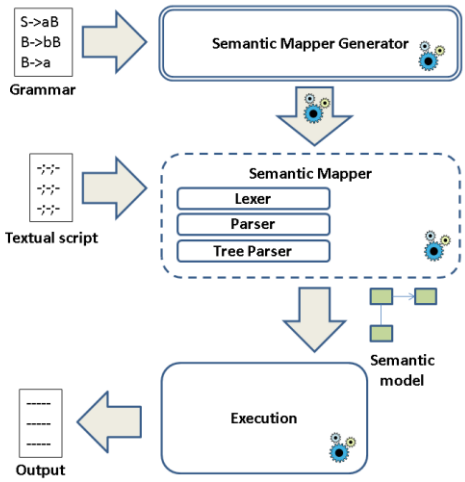


Figure 7
A parser system using a semantic mapper generator

Implementation Most consequences listed in Section 3.5 also apply to semantic mapper generator. Moreover, the grammar must be extended with extra notations that define how language constructs are mapped to the semantic model. Since general parsing algorithms build a syntax tree, designing such a notation has to practically account for the structure of the parse tree. An intuitive implementation idea is used by the Eclipse Xtext [24] project; it maps non-terminals at the left-hand side of parser rules to entities and uses assignment operators, inside the same rules, to assign certain parts of the right-hand side of the rule to properties of the same entity. When these assignments refer to a terminal substitution, a property with a primitive data type will be inferred or when referring to a non-terminal substitution, an association is assumed.

Known Uses The Eclipse Xtext project can be used to create languages. When the language scripts are parsed, it either populates an existing EMF[2] [25] model or infers a new one. This existing or inferred model will be populated with modeling information obtained from the language script. In this case, the known metamodel is EMF; all of the entities, properties and associations are mapped to EMF constructs, which can later be processed with a wide range of tools that work with EMF models.

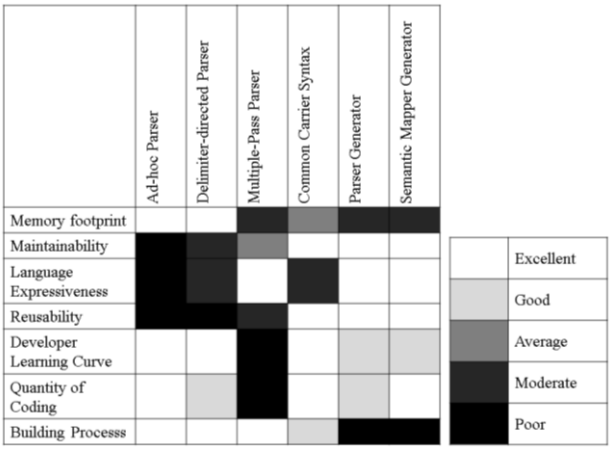
[2] EMF is a metamodeling framework for the Eclipse platform.

Conclusions

In this paper, we have provided a catalog of architectural-level design patterns that can be used to create parsers of computer languages. This catalog summarizes the most important characteristics, motivations and consequences, which can facilitate the engineering work with these kinds of software products. The focus has been limited to the field of parsing, which is a small but important subset of language- and model-processing. Consequently, only architectural-level patterns have been included into the pattern catalog. Table 2 summarizes the most important characteristics of the discussed patterns. We have also shown actors in the industry that are involved in parser development, what their motivations are, and how this pattern catalog can help them to achieve their goals. We believe that the pattern catalog can greatly assist those involved in the industry. Nevertheless, we are also aware that these patterns only help in commencing to design a parser, and do not provide information regarding the more specific details concerning implementation. Potential future work on design patterns for language parsing and modeling could include lower-level best practices in the field of parsing and other common subtasks of model-processing.

Table 2

A summary of the characteristics of the discussed patterns



Acknowledgement

This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred.

References

[1] Fowler, M.: Domain-Specific Languages. Addison-Wesley (2010)

[2] Kelly, S., Tolvanen, J. Domain-Specific Modeling: Enabling Full Code Generation. Wiley - IEEE Computer Society Publications (2008)

[3] van Deursen, A., P. Klint, J. Visser. Domain-specific Languages: an Annotated Bibliography. ACM SIGPLAN Notices. June, 2000

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)

[5] Nguyen, D., Ricken, M., Wong, S.: Design Patterns for Parsing. In: 36th SIGCSE Technical Symposium on Computer Science Education, pp. 477-481, ACM New York (2005)

[6] Schreiner, A. T., Heliotis, J. E.: Design Patterns in Parsing. In: 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181-184, IEEE Press, New York (2001)

[7] Hopcroft, J. E., Motwani, R., Ullman, J. D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (2001)

[8] Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques, & Tools, Second Edition, Addison-Wesley (2007)

[9] Oram, A., Wilson, G.: Beautiful Code: Leading Programmers Explain How They Think. O'Reilly (2007)

[10] Morell, L., Middleton, D.: Recursive-ascent Parsing. Journal of Computing Sciences in Colleges, Volume 18 Issue 6, June 2003, pp 186-201

[11] Roberts, G. H.: Recursive Ascent: an LR Analog to Recursive Descent. ACM SIGPLAN Notices. New York (1988)

[12] Extensible Markup Language (XML) 1.0, http://www.w3.org/TR/xml/

[13] XSL Transformations (XSLT) Version 1.0, http://www.w3.org/TR/xslt

[14] XML Path Language (XPath) Version 1.0, http://www.w3.org/TR/xpath/

[15] XQuery 1.0: An XML Query Language, http://www.w3.org/TR/xquery/

[16] YAML Ain't Markup Language Version 1.2, http://www.yaml.org/spec/1.2/spec.html

[17] Introducing JSON, http://www.json.org/

[18] Mathematical Markup Language (MathML) Version 3.0, http://www.w3.org/TR/MathML3/

[19] Parr, T. J., Quong, R. W.: Adding Semantic and Syntactic Predicates To LL(k): pred-LL(k). CC '94 Proceedings of the 5th International Conference on Compiler Construction. London (1994)

[20] Parr, T. J., Quong, R. W.: ANTLR: a Predicated-LL(k) Parser Generator. Software—Practice & Experience. Volume 25 Issue 7, July 1995, pp. 789-810

[21] Terence, P.: The Definitive ANTLR 4 Reference. Second Revised Edition. Pragmatic Bookshelf (2013)

[22] ANTLR, http://www.antlr.org/

[23] Parr, T., Fisher, K.: LL(*): the Foundation of the ANTLR Parser Generator. PLDI '11 Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. New York (2011)

[24] Eclipse Xtext, http://www.eclipse.org/Xtext/

[25] Eclipse EMF, http://www.eclipse.org/modeling/emf/