

Grammar-Based Systems: Definition and Examples

Marjan Mernik, Matej Črepinšek, Tomaž Kosar, Damijan Rebernak and Viljem Žumer
 University of Maribor, Faculty of Electrical Engineering and Computer Science
 Smetanova ulica 17, 2000 Maribor, Slovenia
 {marjan.mernik, matej.crepinsek, tomaz.kosar, damijan.rebernak, zumer}@uni-mb.si

Keywords: context-free grammars, attribute grammars, grammar-based systems

Received: January 30, 2004

Formal language theory is an important part of theoretical computer science and has also been applied in many practical applications. The importance of context-free grammars and attribute grammars for compiler construction and automatic generation for compilers/interpreters is already well known. However, grammars can be found in many other applications which are not as clearly related to their original application - language description and implementation. We call such systems grammar-based systems. No general comparison and classification has been done until now despite these systems having existed for a long time. The aim of this paper is to introduce and popularize grammar-based systems.

Povzetek: članek opisuje definicije in primere sistemov s formalnimi slovnici.

1 Introduction

This paper emphasizes grammars, especially context-free grammars and attribute grammars. Their importance for compiler construction is already well known. However, from formal language definitions (e.g. attribute grammars) many other language-based tools can be automatically generated [7, 9], such as: pretty printers, syntax-directed editors, type checkers, dataflow analyzers, partial evaluators, debuggers, profilers, test case generators, visualizers, animators, and documentation generators. In most of these cases, the core language definitions have to be augmented with tool-specific information. In other cases, only a part of the formal language definition is sufficient for automatic tool generation, or implicit information must be extracted from the formal language definition in order to automatically generate a tool.

Moreover, grammars can be found in many other applications which are not as clearly related to their original application - language description and implementation. We call such systems grammar-based systems (GBSs). Some papers describing particular approaches even contain this word in their titles (e.g. [4], [33], [36]). However, there is no exact definition nor comparison and classification for such systems. Since grammar-based systems are mainly unnoticed, there is also a lack of identifying benefits of such systems. The aim of this paper is to remedy this situation by defining, introducing and popularizing grammar-based systems. The benefits of GBSs are identified and clearly stated.

The organization of the paper is as follows. Section 2 presents the original application of grammars, namely automatic generation of compilers/interpreters, and other language-based tools using our compiler generator LISA [22]. In section 3, we introduce GBSs, define them and

their application areas, followed by presentation of practical examples in section 4. Concluding remarks and future research work are given in section 5.

2 Original Application of Grammars

The original application of grammars is a notation for language description and its implementation [1]. What do we gain by formalizing the syntax and semantics of a programming language? The following benefits are identified:

- The language definition standardizes the language. This is important to programmers, who need to write syntactically and semantically correct programs and understand them without any doubt about their meaning. It is also important to language implementors, who need to write a correct compiler/interpreter of the specified language.
- The language definition allows a formal analysis of its properties, such as whether the definition is LL(k) grammar and L-attributed grammar. This contributes to better syntax and semantics of the programming language. The programming language that has been formally designed is more regular, has less exceptions and is easier to learn.
- The language definition enables us to systematically derive the implementation of a language, such as a LR(k) parser and attribute evaluator. Moreover, such an implementation can be automatically obtained. In this case, the language definition is used as an input to a compiler generator system. Researchers have

recognized the possibility that many other language-based tools could be generated from a formal language definition. Therefore, many tools not only automatically generate a compiler/interpreter, but also complete language-based environments [7]. Such automatically generated language-based environments include editors, type checkers, debuggers, various analyzers, and animators.

Automatic generation of compilers/interpreters and other language-based tools using our LISA compiler generator are presented in the rest of this section. To support incremental language development [21] and educational activities in teaching “Compiler construction” course [23] the LISA (*Language Implementation System based on Attribute grammars*) tool was developed [22]. LISA is a compiler-compiler, or a system that automatically generates a compiler/interpreter from attribute grammar-based language specifications. The specification of a toy language SELA (Simple Expression Language with Assignments) is given below, in order to illustrate the LISA style.

```
language SELA {
  lexicon {
    Number      [0-9]+
    Identifier  [a-z]+
    Operator    \+ | :=
    ignore      [\x09\x0A\x0D\ ]+
  }

  attributes Hashtable *.inEnv, *.outEnv;
             int *.val;

  rule Start {
    START ::= STMTS compute {
      STMTS.inEnv = new Hashtable();
      START.outEnv = STMTS.outEnv;
    };
  }

  rule Statements {
    STMTS ::= STMT STMTS compute {
      STMT.inEnv = STMTS[0].inEnv;
      STMTS[1].inEnv = STMT.outEnv;
      STMTS[0].outEnv = STMTS[1].outEnv;
    }
    | STMT compute {
      STMT.inEnv = STMTS[0].inEnv;
      STMTS[0].outEnv = STMT.outEnv;
    };
  }

  rule Statement {
    STMT ::= #Identifier \:= EXPR compute {
      EXPR.inEnv = STMT.inEnv;
      STMT.outEnv = put(STMT.inEnv,
        #Identifier.value(), EXPR.val);
    };
  }

  rule Expression {
    EXPR ::= EXPR + EXPR compute {
      EXPR[2].inEnv = EXPR[0].inEnv;
      EXPR[1].inEnv = EXPR[0].inEnv;
      EXPR[0].val = EXPR[1].val +
        EXPR[2].val;
    };
  }

  rule Term1 {
    EXPR ::= #Number compute {
      EXPR.val = Integer.valueOf(
        #Number.value()).intValue();
    };
  }

  rule Term2 {
    EXPR ::= #Identifier compute {
      EXPR.val = ((Integer)EXPR.inEnv.get(
        #Identifier.value())).intValue();
    };
  }
}
```

```
}
}
```

LISA automatically generates a SELA compiler/interpreter from this specification. An example of a program written in the SELA language is shown in Figure 1.

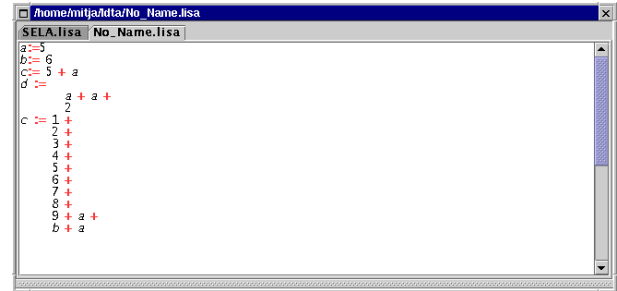


Figure 1: Language knowledgeable editor

LISA also automatically generates other tools, such as language knowledgeable editors and various inspectors (e.g. finite state automata visualizer (Figure 2), syntax and semantic tree animators (Figure 3)) that are useful for understanding the behavior of the generated language compiler/interpreter. A LISA-generated language knowledgeable editor is aware of the regular definitions of the language lexicon. Therefore, it can color the different parts of a program (comments, operators, reserved words) to enhance the understandability and readability of programs. In Figure 1 the operators in the SELA program are recognized while editing and displaying in a different color.

3 Grammar-Based Systems: Definition

As already mentioned, grammars can be found in many other systems than those described in section 2. These systems do not focus on language definition and implementation, but on solving various other problems. We call such systems grammar-based. The essential characteristics of GBSs is comprised in the following definition:

A grammar-based system is any system that uses a grammar and/or sentences produced by this grammar to solve various problems outside the domain of programming language definition and its implementation. The vital component of such a system is well structured and expressed with a grammar or with sentences produced by this grammar in an explicit or implicit manners.

The key characteristic of GBSs according to our definition is a grammar which presents a kernel for the problem solving part of the system (application). Without this grammar part, the system becomes less general, and by lacking an important generic functionality, it can become usable

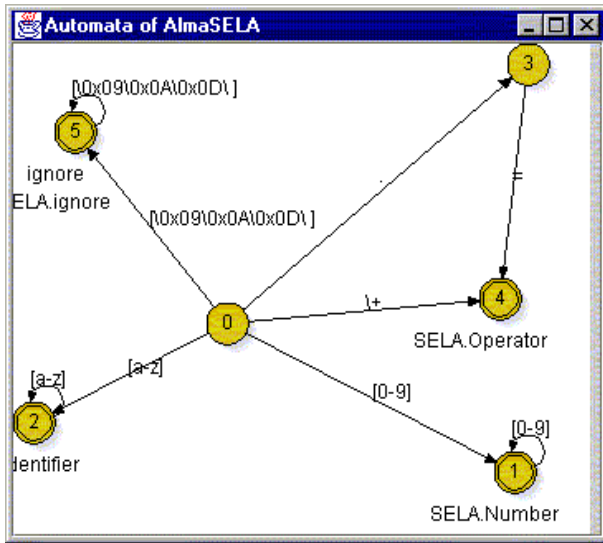


Figure 2: FSA visualizator

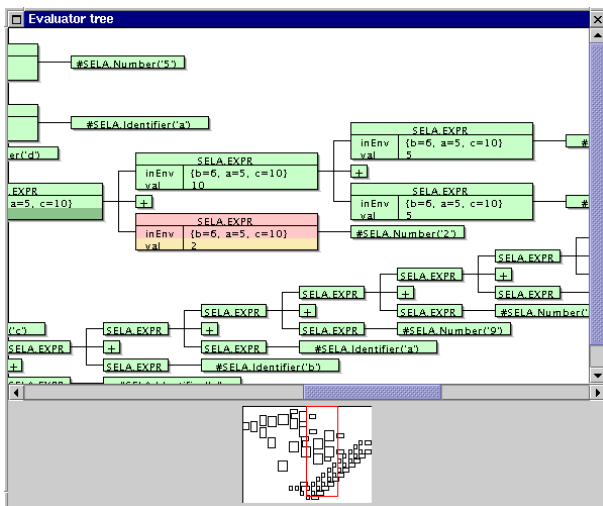


Figure 3: The snapshot of semantic tree animator

only in a very restricted manner. In many GBSs the transformation of the representation to a context-free grammar makes the various analyses of properties feasible. In others the ability of context-free grammars to represent infinite languages with a finite set of production rules is exploited. Sometimes, a problem can be solved simply by converting the representation to a context-free grammar since the appropriate tools already exist. Why is the study of GBSs so important? The theory of grammars is well-defined and described in many books, where different examples and solutions are presented [1, 32]. In our research we discovered that grammars can be, and already are, used as a kernel part of many different practical systems. The fact that grammars are so well-defined is an advantage for developers, because they can use that knowledge, and the already well tested solutions, to build their own. The main problem of using grammars as part of a solution is to identify those problems that have grammatical nature (can be solved with grammars) and to convert their presentation in the form of grammars.

One may ask why traditional programming applications such as compilers are excluded from the above definition since it is clear that these applications heavily depend on grammars. Actually, their whole construction is based on grammars. Grammars have been used in this area since their invention and other ad-hoc approaches were mainly superseded by grammars a long time ago. In this case we simply do not have other options. Therefore, talking about grammar-based compiler would be awkward. On the other hand, using grammars in other application areas can be regarded as an alternative and novel approach with clearly defined benefits. In this case the noun qualifier “grammar-based” is really appropriate.

Our longstanding interest in grammars inspired us to start collecting information about their different practical application areas, such as:

- *Software engineering*, where syntax definition occurs in various software development processes – in the form of rapid prototyping [28, 8], the modeling flow and constrains of collaborating software components [17], and many others [3, 15].
- *Evolutionary computation* is the study of computational systems that uses ideas from natural evolution and adoption to search the solution space. One of the research fields of evolutionary computations is grammatical evolution [26].
- *Information theory* comprises a vast range of diverse scopes. So far, our observations noticed grammars involved in encoding methods [2, 25], programming compaction [4] and grammatical inference [19]. Other grammar-dependent software in information theory are under investigation.
- *Neural networks* bring grammars into use with grammatical description of neural networks topology [14, 6, 10].

- *Data representation* architecture uses special kinds of grammars for communicating business data among very diverse systems. The particular technology considered is XML [29].
- Other areas of Computer Science (speech recognition, data mining, syntactical pattern recognition, etc).

Applications of grammars are found even in areas outside computer science, such as organizational science [27] and mechanical engineering [33].

Until now GBSs have been studied only for particular problems (e.g. compression) without any general comparison and classification. The only attempt, to the best of our knowledge, is a recent work described in [15] where authors coined the word *grammarware*. To quote their definition “*Grammarware comprises grammars and all grammar-dependent software, i.e. software artifacts that directly involve grammar knowledge.*” Their definition classified compilers, program analysis tools, program transformation tools, application generators, weaving tools, CASE tools as grammarware. Their definition, is in some, sense more restricted than ours and includes just GBSs from those areas of software engineering where grammars appear in an explicit form. Sentences produced by this grammar are always computer programs. Our definition is much broader since we are interested in GBSs for application areas that are even outside computer science (e.g. organizational science). On the other hand, in our definition grammars and sentences produced by this grammar can be expressed implicitly (e.g. in GOOD [17] a sentence is a sequence of method calls during the execution of an application program).

4 Grammar-Based Systems: Examples

GBSs can be found in different application areas such as: software engineering, evolutionary computations, information theory, neural networks, data mining, syntactical pattern recognition, and data representation. In this section some of our own applications, as well as other representative applications of grammars, are presented and their benefits are stressed in more detail. Examples clearly show how grammars and sentences generated by a grammar can be used to describe various structured artifacts. Moreover, various possibilities exist for using GBSs. Examples include descriptions of GBSs where sentences generated by grammar appear in explicit or implicit manner.

4.1 Software Engineering

4.1.1 Grammatical Approach to Problem Solving

In [8, 28] the grammatical approach to problem solving (GAPS) is presented. It is based on the following steps:

- describe the syntax of the problem (the structure of the classes that characterize problem domain), deriving the context-free grammar from the conceptual class diagram,
- describe the semantics of the problem (the meaning of the classes in the problem domain), associating attributes to every concept derived from the use cases and operational diagrams,
- generate a rapid prototype of the system, using a compiler generator and the attribute grammar obtained in the two previous steps.

Only the first step is explained for the purpose of this paper. A detailed explanation of the above steps can be found in [8].

The role of non-terminal symbols in a context-free grammar is two fold. First, at a higher abstraction level non-terminal symbols are used to describe different concepts in the programming language (e.g. an expression or a declaration in a general-purpose programming language). On the other hand, at a more concrete level, non-terminal and terminal symbols are used to describe the structure of a concept (e.g. an expression consists on two operands separated by an operator symbol, or a variable declaration consists of a variable type and a variable name). Therefore, both the concepts and the relations between them, belonging to the specific problem domain, are captured in a context-free grammar. But, this is also true for the conceptual class diagram [30] which describes concepts in a problem domain and their relations. It is clear that both formalisms can be used for the same purpose and that some rough transformation from a conceptual class diagram to a context-free grammar and vice versa should exist. The transformation from a conceptual class diagram to a context-free grammar is depicted in Tables 1 and 2.

Classes can collaborate with more than just one class. For example, class A associates with classes B, C and D. In our approach, this collaboration is described with context-free grammar production $A \rightarrow B C D$. The sequence of non-terminal symbols on the right side of the production should be in natural order and depends on the collaboration of entities in a given problem domain.

As an example let’s transform the conceptual class diagram in Fig. 4 to a context-free grammar. From this conceptual class diagram the following context-free grammar is obtained using transformation Tables 1 and 2.

```
VIDEO_STORE ::= MOVIES CUSTOMERS
MOVIES      ::= MOVIES MOVIE | MOVIE
MOVIE       ::= title PRICE
CUSTOMERS   ::= CUSTOMERS CUSTOMER | epsilon
CUSTOMER    ::= name RENTALS
RENTALS     ::= RENTALS RENTAL | RENTAL
RENTAL      ::= daysRented MOVIE
PRICE       ::= new | child | reg
```

The next step of the grammatical approach is to write a detailed semantic description of the problem domain deriving the attribute grammar. The prototype of the system is




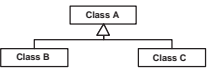


Association	Class diagram element	Grammar
Class		Class (non-terminal) attribute (terminal)
Association		$A ::= B$
Navigability		$A ::= B$
Generalization		$A ::= B \mid C$
Aggregation		$A ::= B$ $(\neg \exists X \in N, X \Rightarrow B)$ $\wedge X \neq A$
Composition		$A ::= B$

Table 1: From a conceptual class diagram to a context-free grammar

obtained after a straightforward transformation of attribute grammar specification to LISA specification. The following program describes a particular use of the system.

```
//entering movies into database
lion_king child
gone_with_the_wind reg
the_ring new
//customers and their rentals description
Andy 3 lion_king child
    2 gone_with_the_wind reg
Mary 3 the_ring new
```

What are the advantages of using grammars in this case? By transforming the conceptual class diagram to a context-free grammar a domain-specific language is obtained that describes the user interaction with the system. In this manner a rapid prototype is obtained and can be used whenever





Cardinality	Class diagram element	Grammar
Multiplicity exactly one		$A ::= B$
Optional multiplicity		$A ::= B \mid \epsilon$
Multiplicity [0..m]		$A ::= \text{MoreB}$ $\text{MoreB} ::= \text{MoreB B} \mid \epsilon$
Multiplicity many		$A ::= \text{MoreB}$ $\text{MoreB} ::= \text{MoreB B} \mid B$

Table 2: Association multiplicity

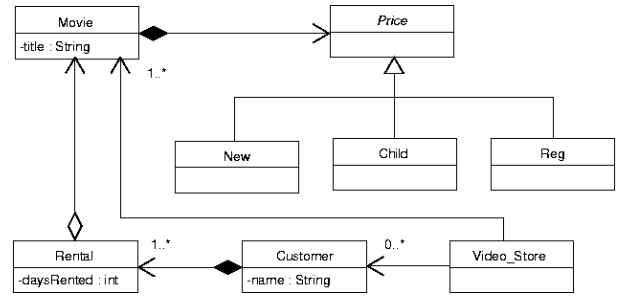


Figure 4: Conceptual Class Diagram for Video Store

the user’s requirements are not well defined. Another benefit is that a conformability check of the conceptual class diagram is also possible.

4.1.2 Adaptive programming

Adaptive programming (AP) [18] is a subclass of aspect-oriented programming [13]. The stress is put on code tangling – for example, the required functionality is not always trivial to implement in existing applications when cross-cutting concerns exist. Adaptive programming offers a solution in the form of traversal specifications, in order to provide that additional functionality without modification of the existing code. These specify connections between objects as loosely as possible (called “structure-shy” programming).

The structure of the application class dictionary can be seen as a context-free grammar (Figure 5) from which an object graph may be derived, express all possible navigations through the code.



Figure 5: Class dictionary (CFG) and its object graph

The idea of adaptive programming is very general. The Demeter [18] language has been integrated with various object-oriented programming languages (the Demeter tool was successfully applied to Java). Demeter allows programmers to write the following specifications (see Fig. 5):

```
starting from object A, go to object C
via all objects with an attribute named "x".
```

to add arbitrary execution paths.

The problem of adaptive programming can also be solved using other techniques (visitor pattern). In comparison, the presented solution avoids code tangling, increases the programmer’s productivity and consequently, it reduces error prone coding.

4.1.3 Other Approaches

In Grammar-Oriented Object Design (GOOD) [17] a context-free grammar is used to represent a set of all possible interactions (collaborations) for objects in a particular

cluster, in order to fulfill the domain goals. When a grammar is interpreted at run-time, a cluster will dynamically bind the collaborators to the collaborations. Hence, GOOD facilitates the creation of dynamically configurable components, which encapsulates volatile business rules. The rationale behind this is that creating and representing a model of solutions is more extensible, simpler and more scalable than just creating the single solution. Possible solutions are modeled with a meta-model and represented as a context-free grammar. If this grammar is available to the “users” at run-time, then they are able to customize the system’s behavior. An example of a production rule in [17] using EBNF is:

```
ShoppingCartOperation ::=
  {AddItem | DeleteItem |
   SaveShoppingCart} CheckOut
```

Since the interaction of objects is obtained from use case diagrams that describe the functionality of a system, the author [17] called such a grammar a use case grammar. The author [17] in his work distinguishes two types of meta-models: the static (class diagram) and the dynamic (valid object interaction sequences) meta-model. The latter is described with a context-free grammar.

In [3] the correspondence between the feature diagram and the context-free grammar has been identified, where atomic features map to terminal symbols, composite features map to nonterminal symbols, and feature operators map to syntax operators. In domain analysis, feature diagrams are used to describe commonalities, variabilities and dependencies between variable properties in the application domain. By converting a feature diagram to a context-free grammar (FD2CFG), syntax tools can be applied to feature descriptions for free (e.g. validity of configuration corresponds to successful parsing).

The Free University of Amsterdam recently launched a project on Grammar Engineering - software engineering for grammars [15]. Topics included are grammar recovery, grammar implementation, and the application of grammars in software renovation. The more technical issues include concepts and technology for grammar-based software renovation factories, grammar adaptation, grammar documentation, grammar testing and many others.

4.2 Evolutionary Computations

Genetic programming (GP) is an evolutionary approach in which an evolving population consists of computer programs [16]. Each member of the population, a chromosome, represents a possible solution in the search space of all possible programs written in a *pre-selected programming language* (e.g. Lisp). Since the search space is too large it is restricted by the user-defined function set F and the terminal set T . The set T contains variables and constants and the set F functions that are a priori believed to be useful for the problem domain. For example, in the Santa Fe ant trail problem [16] from sets

$T = \{(\text{MOVE}), (\text{LEFT}), (\text{RIGHT})\}$ and $F = \{\text{IF-FOOD-AHEAD}, \text{PROGN2}, \text{PROGN3}\}$ the following solution (lisp program) can be evolved:

```
(IF-FOOD-AHEAD (MOVE) (PROGN3 (LEFT) (PROGN2
(IF-FOOD-AHEAD (MOVE) (RIGHT)) (PROGN2 (RIGHT)
(PROGN2 (LEFT) (RIGHT)))) (PROGN2
(IF-FOOD-AHEAD (MOVE) (LEFT)) (MOVE))))
```

In [26] the concept of grammatical evolution (GE) has been introduced. GE is an evolutionary algorithm that can evolve programs in an *arbitrary language* [31]. The input to the GE is a BNF definition for the genotype-to-phenotype mapping process. For example, the following grammar can be used as input to Santa Fe ant trail problem:

```
0. CODE ::= LINE
1. CODE ::= CODE LINE
0. LINE ::= EXPR
0. EXPR ::= IF-STAT
1. EXPR ::= OP
0. IF-STAT ::= if (food-ahead()) EXPR else EXPR
0. OP ::= left()
1. OP ::= right()
2. OP ::= move()
```

The population consists of variable-length binary strings that determine which production rules from the grammar definition are used in a genotype-to-phenotype mapping process. The appropriate production rule is selected by using the following mapping function:

$$rule = (\text{Integer value stored in a chromosome}) \text{MOD} (\text{number of production rules for the left most nonterminal})$$

For example, the following chromosome (203 245 110 55 29 200 241 11 151 162 227 74) encodes the following left-most derivation

$$\begin{aligned} CODE &\Rightarrow^{203 \text{MOD} 2} CODE \quad LINE \Rightarrow^{245 \text{MOD} 2} \\ CODE \quad LINE \quad LINE &\Rightarrow^{110 \text{MOD} 2} LINE \quad LINE \quad LINE \Rightarrow \\ EXPR \quad LINE \quad LINE &\Rightarrow^{55 \text{MOD} 2} OP \quad LINE \quad LINE \Rightarrow^{29 \text{MOD} 3} \\ move() \quad LINE \quad LINE &\Rightarrow \dots \Rightarrow \\ move() \quad if(\text{food-ahead}()) \quad move() \quad else \quad left() \quad move() &\end{aligned}$$

What are the benefits of using grammars in GE? Obviously, GE is much more flexible than GP because it can produce a code in any language. Furthermore, in the GE closure problem, the generation and preservation of valid programs, does not exist. Other benefits come with the separation of the search and solution spaces because grammar enables the genotype-to-phenotype mapping process. This allows an unconstrained evolutionary search to be performed on simple variable-length binary strings. Moreover, new advances in genetic algorithms can be easily incorporated into GE or any new search algorithm operating on binary strings can be used.

4.3 Information Theory

The ability of a grammar to represent an infinite language with a finite set of production rules also makes grammars useful in compression algorithms. Grammar-based encoding (GBEnc) methods, such as derivation encoding [34], which represents a program by a sequence of grammar rules to derive it from the start symbol, have been proven

useful for compressing programs. For example, the program using the grammar from subsection 4.2 can be encoded as 110120121012 by derivation encoding. The derivation tree is shown in Figure 6. It was shown in [2] that programs can be compressed to almost 10% of their original size. Another grammar-based compression algorithm is SEQUITUR [25] which constructs a context-free grammar for its input. The resulting grammar is capable of generating just one string, namely the original sequence. For example, the sequence *abcdbcabcd* is represented by the following grammar

S ::= CAC
 A ::= bc
 C ::= aAd

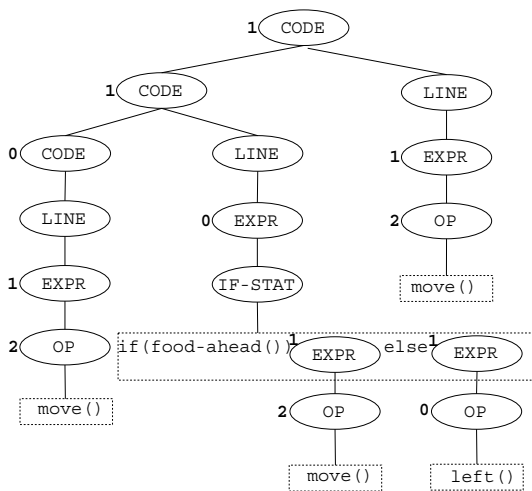


Figure 6: Derivation tree

The algorithm identifies the hierarchical structure (Figure 7) in sequences of symbols and uses that information for compression. By detection and elimination of redundancy it outperforms the standard compression techniques on very large or highly structured sequences. SEQUITUR performs well on many practical problems such as DNA sequences and genealogical databases. It is also possible to use the system as a basis for generalization in grammatical inference [19]. Grammar-based techniques (e.g. [4]) have also been used in program compaction, which is a compression technique with an additional constraint - the compressed program has to be executable.

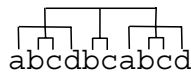


Figure 7: Hierarchical structure for grammar

4.4 Neural Networks

The selection of a suitable neural network topology is an important step in finding a good solution for the problem

under investigation. Therefore, the search for suitable neural network architecture is a common task. Here we can use direct encoding, or the so-called grammatical encoding [14], where the architecture of the neural network is generated from its grammar description. The advantages of grammatical encoding are better scalability and the possibility of finding building blocks. This work is further elaborated in [6] where cellular encoding using graph grammars was proposed. The architecture, the weights, and the kind of sigmoids used by each neuron are encoded. Cellular encoding can be seen as a machine language for neural networks and can be used as a tool for designing neural networks.

In [10] an attribute grammar is used to specify classes of neural network structures with explicit representation of their functional organization. The approach is termed Network Generating Attribute Grammar Encoding (NGAGE). The specification of a neural network structure is extracted from the attributes of the root symbol and interpreted to produce a functional neural network. This neural network can be randomly initialized and trained afterwards. The NGAGE is specially usable in genetic programming, in the form of neural network representation, where each production rule (derivation subtree) corresponds to a meaningful structural component of the neural network. These characteristics of NGAGE can be used for genetic operators implementation, crossover and mutation. The other benefit of NGAGE is identical representation of different neural networks. The similarities and differences between them can be emphasized within a common framework.

4.5 Data Representation

The use of mark-up languages on the Web is indispensable. The *hypertext mark-up language* (HTML) is the best known example. In the last few years the *eXtensible Markup Language* (XML) has been introduced, as a mark-up language for uniform representation of data. It was originally meant as a format for transferring data over the Internet. Separation of data from its representation increased the standard applicability to other computing areas.

Although our perception is that compiler notation and mark-up language have little in common, the reality is quite different. The syntax of XML documents is conceptually similar to the meta-language (defined by BNF) of compilers. The analogy between compilers and mark-ups is shown in table 3.

Notation	Compiler	Mark-up
meta-notation	LISA, ASF+SDF	XML
syntax	context-free grammar	DTD, XML Schema

Table 3: Analogy between compiler and mark-up

The syntax of an XML document is defined by Document Type Definition (DTD). DTD defines document structure, elements, their attributes and types (see example be-

low). It uses the syntax of EBNF ('*', '+', '?', '|') to describe the syntactical structure of XML documents. Therefore, a DTD can be seen as an extended BNF (EBNF).

```
<!ELEMENT paper_collection (paper)*>
<!ELEMENT paper (title, author+, year,
                 published?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT published EMPTY>
```

An example of an XML document, for the given DTD above, is written below. The XML elements from the DTD can be seen as non-terminal symbols in EBNF. *Empty elements* are without contents (element '<published>') - their meaning is in the position and attributes. *Non-empty elements* can contain other elements and textual content (element '<title>'). Textual content can be parallelized with terminal symbols in EBNF.

```
<paper_collection>
  <paper>
    <title>Grammatical Approach to
      Problem Solving</title>
    <author>Pedro Henriques</author>
    ...
    <year>2003</year>
    <published/>
  </paper>
  ...
</paper_collection>
```

Another similarity of mark-up languages and compilers is the building of a parser. The XML parser generator analyzes the source DTD and automatically generates a parser for XML documents which comply with the source DTD, which is also the case in compiler building tools (e.g. LISA [20]).

The disadvantage of XML against context-free grammar is in its lexical part. The textual content of XML elements can be either a generic string (denoted with '#PCDATA' in DTD) or enumeration of allowed values. This limitation is reduced with XML Schema, which offers richer notation for describing the textual content of the XML elements.

4.6 Other Applications

Due to page limitation, all GBSs can not be described in detail. The aim of the paper is to show a plethora of different research areas where grammars are proven to be useful. A short description of such systems in other areas follows.

A number of researchers have proposed ways to use grammar-based notation for expressing knowledge in the speech recognition process. In most cases CFG was used to generate or filter word transitions [24]. To improve semantic sentence recognition, the probabilistic LR parser has been used as well as stochastic CFG (SCFG) [12].

Data mining is an automated process of discovering knowledge from databases. Various data mining methods exist, among them are inductive logic programming and

genetic programming. In [36] both approaches were integrated using logic grammars aiming to exploit the benefits of both approaches. Special rule learning has been developed where a grammar represents rules. Moreover, the grammar can be modified in order to learn rules.

Formal language theory has been successfully applied to pattern recognition problems [5] in which the patterns contain most of their information in their structure rather than in their numeric values. In order to make grammars more suitable for pattern recognition the concept of context-free grammars have been extended to stochastic grammars [35] and fuzzy grammars [11].

The design of organizational and work processes is well defined. However, there is a lack of formal approaches in discovering new organizational processes. By using grammars, one may systematically search for solutions in process redesign, as well as for new solutions in process organization. Therefore, the process grammar [27] offers complementary solutions to the existing ones.

4.7 Concluding remarks on GBSs examples

It is important that GBSs are not studied in an isolated manner. In order to be able to make a general comparison of different GBSs and to classify them, we need to find their common and variable properties. The main reason for classification of GBSs is to identify differences among GBSs and to identify the representative examples of it. These classifications can be used by future developers to identify whether their system is solvable using a grammar-based approach. Developers can further use the classification to build their own system faster and more efficiently.

The following questions help to identify different dimensions of GBSs:

- Q1 What is described with grammar G?
- Q2 What is described with program P generated by language L(G)?
- Q3 Is the representation of program P generated by language L(G) explicit?
- Q4 Is the control flow from $G \rightarrow P$ or $P \rightarrow G$? (Is the input to GBS defined by grammar or program?)
- Q5 Why was GBS invented?

In table 4, the answers to some examples are given.

It is important for future application developers to notice that with a grammar-based approach their systems can benefit in several directions:

- system can become more general (e.g. GOOD [17]),
- system can be easier to develop (e.g. [3]),
- system's underlying representation can be more efficient (e.g. [10]).

	Q1	Q2	Q3	Q4	Q5
GAPS	conceptual class diagram (CCD)	user interaction with the system	yes	$G \rightarrow P$	to obtain rapid prototype of the system
GOOD	interaction between objects	sequence of method calls executed by an application program	no	$G \rightarrow P$	to extend generality of the system
FD2CFG	feature diagram (FD)	an instance of a system described by FD	yes	$G \rightarrow P$	to check if an instance is a valid system by FD
GE	grammar of the target language used in GP	program written in a target language	no	$G \rightarrow P$	to extend generality of the system
GBEnc	grammar of the target language	program written in a target language to be compressed	yes	$P \rightarrow G$	for compression
SEQUITUR	grammar of the target sequence of symbols	sequence of symbols	yes	$P \rightarrow G$	for compression
AP	connections between objects	structure of application class dictionary	no	$G \rightarrow P$	to extend functionality of applications
NGAGE	neural network (NN) structure	a fully functional NN	no	$G \rightarrow P$	to simplify NN representation for GP

Table 4: A comparison among different GBSs

This paper describes some of the representative examples of the above mentioned benefits. The main contribution of this paper therefore is:

- definition of grammar-based systems,
- identifying problems that can be solved with grammar-based approach,
- identifying benefits of grammar-based system, and
- popularizing grammar-based systems.

5 Conclusions and future work

Formal language theory has been applied to many practical applications. In addition to language description and implementation (original applications of grammars), grammars have been proven useful in many other areas. However, there is no particular research of systems (applications) in which grammar plays a vital role. In this paper such systems are introduced and defined as GBSs. The paper contains representative examples of GBSs in various areas of computer science, such as: software engineering, evolutionary computations, information theory, neural networks, and data representation.

Although the formal theory of grammar is well defined, there are still many research possibilities in the field of GBSs. We have noticed several unexplored areas in GBSs such as:

- *Classification of GBSs.* There is no classification of GBSs. We believe that this can be attained by questions similar to the ones proposed in section 4.7. However, further case studies of GBSs are required.
- *When to develop GBS?* No guidelines exist to show whether a particular problem should be solved with grammar knowledge.

- *GBSs patterns.* The remaining question is how to develop GBSs. Identifying patterns would improve and speed up the interest in developing GBSs.

In the future we plan to extend our research on GBSs. Our research will be focused on solving those problems presented in this paper and on finding other areas or problems that can be efficiently solved with the grammar-based approach. We want to show that problem definition using the formal approach (grammar) can increase the efficiency, reliability and generality of the solution.

6 Acknowledgements

We would like to thank Jeff Gray and anonymous referees for useful comments.

References

- [1] A. V. Aho and J. D. Ullman. The theory of languages. *Mathematical Systems Theory*, 2(2):97–125, 1968.
- [2] R. Cameron. Source encoding using syntactic information models. *IEEE Transactions on Information Theory*, 34(4):843–850, 1988.
- [3] M. de Jonge and J. Visser. Grammars as feature diagrams. draft, Apr. 2002.
- [4] W. S. Evans and C. W. Fraser. Grammar-based compression of interpreted code. *ACM Communications*, 46(8):61–66, 2003.
- [5] K. Fu. *Syntactic Pattern Recognition and Applications*. Prentice-Hall, 1982.
- [6] F. Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm*. PhD thesis,

- Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, France, 1994.
- [7] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM Sigplan Notices*, 35(3):39–48, Mar. 2000.
- [8] P. Henriques, T. Kosar, M. Mernik, M. J. V. Pereira, and V. Žumer. Grammatical approach to problem solving. In *ITI 2003 : Proceedings of the 25th International Conference on Information Technology Interfaces*, pages 645–650. SRCE University Computing Centre, University of Zagreb, 2003.
- [9] P. Henriques, M. V. Pereira, M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Automatic generation of language-based tools. In M. van den Brand and R. Laemmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.
- [10] T. Hussain and R. Browse. Attribute grammars for genetic representations of neural networks and syntactic constraints of genetic programming. In *AIVIGI'98: Workshop on Evolutionary Computation*, 1998.
- [11] Y. Inagaki and T. Fukumura. On the description of fuzzy meaning of context-free languages. In L. Zadeh, K. Fu, K. Tanaka, and M. Shimura, editors, *Fuzzy Sets and Their Applications to Cognitive and Decision Processes*, pages 301–328. Academic Press, 1975.
- [12] D. Jurafsky, C. Wooters, J. Segal, A. Stolcke, E. Foslner, G. Tajchman, and N. Morgan. Using a stochastic context-free grammar as a language model for speech recognition. In *Proc. ICASSP '95*, pages 189–192, Detroit, MI, 1995.
- [13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.
- [14] H. Kitano. Designing neural networks by genetic algorithms using graph generation systems. *Complex Systems*, (4):461–476, 1990.
- [15] P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. Draft, Aug. 2003.
- [16] J. R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, 1992.
- [17] K. Levi and A. Arsanjani. A goal-driven approach to enterprise component identification and specification. *Communications of the ACM*, 45(10):45–52, October 2002.
- [18] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.
- [19] M. Mernik, M. Črepinšek, G. Gerlič, V. Žumer, B. R. Bryant, and A. Sprague. Learning context-free grammars using an evolutionary approach. Technical report, University of Maribor and The University of Alabama at Birmingham, 2003.
- [20] M. Mernik, N. Korbar, and V. Žumer. LISA: A tool for automatic language implementation. *ACM SIGPLAN Notices*, 30(4):71–79, Apr. 1995.
- [21] M. Mernik, M. Lenič, Enis Avdičaušević, and V. Žumer. Multiple Attribute Grammar Inheritance. *Informatica*, 24(3):319–328, Sept. 2000.
- [22] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. LISA: An Interactive Environment for Programming Language Development. In N. Horspool, editor, *11th International Conference on Compiler Construction*, volume 2304, pages 1–4. Lecture Notes in Computer Science, Springer-Verlag, 2002.
- [23] M. Mernik and V. Žumer. An educational tool for teaching compiler construction. *IEEE Transactions on Education*, 46(1):61–68, February 2003.
- [24] R. Moore, F. Pereira, and H. Murveit. Integrating speech and natural-language processing. In *Proc. of the Speech and Natural Language Workshop*, pages 243–247, Philadelphia, PA, 1989.
- [25] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40:103–116, 1997.
- [26] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transaction on Evolutionary Computations*, 5(4):349–358, August 2001.
- [27] B. T. Pentland. Grammatical models of organizational processes. *Organization Science*, 6(5):541–56, 1995.
- [28] M. V. Pereira, M. Mernik, T. Kosar, P. Henriques, and V. Žumer. Object-oriented attribute grammar based grammatical approach to problem specification. Technical report, University of Braga, Department of Computer Science, 2002.
- [29] E. T. Ray. *Learning XML: Creating Self-Describing Data*. O'Reilly & Associates, Inc., 2001.
- [30] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
- [31] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 83–95, Paris, 14–15 Apr. 1998.

- [32] Salomaa, A. *Theory of Automata*. Pergamon Press, 1969.
- [33] L. C. Schmidt and J. Cagan. Ggreada: A graph grammar-based machine design algorithm. *Research in Engineering Design*, 9:195–213, 1997.
- [34] R. G. Stone. On the choice of grammar and parser for the compact analytical encoding of programs. *The Computer Journal*, 29(5):307–314, 1986.
- [35] P. Swain and K. Fu. Stochastic programmed grammars for syntactic pattern recognition. *Pattern Recognition*, (4):83–100, 1972.
- [36] M. L. Wong and K. S. Leung. *Data mining using grammar based genetic programming and applications*. Kluwer Academic Publishers, 2000.