# Web Service for designing and implementing formal languages

Damijan Rebernak, Matej Črepinšek, Marjan Mernik
University of Maribor, Faculty of Electrical Engineering and Computer Science
Smetanova ulica 17, 2000 Maribor, Slovenia
*{damijan.rebernak, matej.crepinsek, marjan.mernik}@uni-mb.si*
tel: ++386-2-220-7462 fax: ++386-2-2511-178

**Abstract.** *An intelligent web service and its interface for designing and implementing formal languages is described in the paper. The paper describes also background tools LISA and GIE used in the web service and some basics of their usage. Three language design approaches are presented. Firstly, from scratch using LISA specifications, secondly using stored specifications from other developers, and thirdly with a set of positive and negative example strings/programs. Advanced user interface with shared and documented repository of solutions make web service suitable for beginners as well as compiler experts. Web service has been designed in a way which makes it suitable for integration in standalone applications.*

**Keywords.** Attribute grammars, compiler/interpreter generator, language design, grammar inference

## 1 Introduction

Despite the fact that compiler construction (particularly parsing) is one of the most understood branches of computer science, language design and its implementation is still considered a very difficult task. We believe that one of the most difficult tasks in designing a language is writing its semantics/grammar, to make language formal. Grammars are already being used in many disciplines: in computer science (for compiler construction, database interfaces, artificial intelligence, etc.), in linguistics (for text analysis, machine translation), document preparation, etc. We also noticed that grammars are suitable for many other applications which are not closely related to their originating area - language description and implementation. We call such systems grammar-based systems (GBSs) [6]. Necessity for that kind of knowledge is present in almost every industry and academic branch. To increase the overall efficiency of language development many tools for automatic language generation from formal specifications have been developed. Tools like Yacc, Lex [5], ASF+SDF [12], and LISA [8] can decrease development time dramatically, but are still relatively difficult to use. The main drawback of these tools is their inappropriateness for users that are not language design experts.

In the paper the interface for our web service for designing and implementing formal languages (grammar/scanner/parser/evaluator) is presented. The intelligent web service (web interface available at – *http://www.cs.feri.uni-mb.si*) is composed of two different software components. The first is LISA [8] (***L**anguage **I**mplementation **S**ystem based on **A**ttribute grammars*)[1] and the second is GIE[2] [2] (***G**rammar **I**nference **E**ngine*)[3]. Why do we need yet another compiler generator tool? Our web service is not an ordinary tool for designing languages. A grammar (compiler or just scanner/parser) can be designed and generated in three ways. We can design it from scratch using LISA specifications. The other possibility is to browse the repository of already implemented grammars by other developers and extend them using multiple attribute grammar inheritance [7] to get the desired solution. The third option is to start with examples (positive and negative) of source strings/programs. Web service will then guide the developer through the process of gen-

[1] *http://marx.uni-mb.si:8080/LisaWebService/services/CServiceBean?wsdl*

[2] This work is sponsored by bilateral project BI-US/03-04/5 "GenParse: Generating a Parser from Examples" between Slovenia and USA.

[3] *http://marx.uni-mb.si:8080/GIEWebModule/services/GIEWebServiceBean?wsdl*

erating a parser from a set of positive and negative examples using grammatical inference. The generated parser can be used as a standalone unit or can be transformed into LISA specifications and then further extended with semantic rules.
The web service is appropriate both for beginners as well as experts. Beginners can use repository of example specifications to start designing their own language. Experts can use web service functionalities in their own applications.
The organization of the paper is as follows. Section 2 gives a short overview of web service background tools LISA and GIE. In section 3 we present the core of web service including all its features. Finally, the concluding remarks and some ideas for additional work are presented in section 4.

# 2 Background Tools

As already mentioned in the introductory part, the web service is based upon two different tools – LISA and GIE. A short overview of those tools is presented in the following two subsections. Detailed information can be found in [1, 2, 8].

## 2.1 Lisa ver. 2.0

The LISA tool is an interactive environment for the development of programming languages [8]. LISA takes formal attribute grammar-based specifications as input and produces a compiler/interpreter for a programming language defined by those specifications. The generated compiler is a source Java file. 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     [\0x09\0x0A\0x0D\ ]+
 }

 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();
   };
 }
}
```

As can be observed from the presented example, the specifications of a programming language in LISA consist of lexical (regular definitions), syntactical (production rules – BNF form) and attribute grammar semantic specifications. LISA supports component–oriented incremental language development based on multiple attribute grammar inheritance [7]. Attribute grammar inheritance is a very effective language design mechanism as it allows new definitions to be built upon existing ones. A new specification inherits the properties (regular definitions, attributes, production rules and semantic rules) of its ancestors and may introduce new properties that extend, modify or override its inherited properties.
LISA also generates other tools, such as language knowledgeable editors and various inspectors such as finite state automata visualizator and syntax and semantic tree animators that are useful for understanding the behavior of the generated language.

## 2.2 Learning grammar from examples (GIE – tool)

Using tools like LISA simplifies work with grammars, but it is still a very complex and long–term process for domain experts to build a programming language. They usually know their inputs and outputs very precisely, but describing that data formally is difficult for them. Idea of a system that will find structure description from examples arises. The best way to accomplish this is by using techniques from grammatical inference (grammar induction or language learning).

### 2.2.1 Grammatical Inference

Grammatical inference, a subarea of machine learning, is a process of learning grammars from training sets. One of the most important theoretical aspects in this field is Gold's theorem [3] which states that it is impossible to identify any of the four classes of grammars in the Chomsky hierarchy when using only positive examples. Using both negative and positive examples, the Chomsky hierarchy of grammars can be identified. Using only positive examples results in an uncertainty as to when the generalization steps should be stopped. This implies the need for some restrictions or background knowledge on the generalization process. Despite the disappointing results, research has continued in this area. Learning algorithms have been developed that exploit knowledge of negative examples, structural information, and are capable of restricting grammars to some subclasses (e.g. even linear grammars, k-bounded grammars, structurally reversible languages, terminal distinguishable context-free languages, etc.) [4] in which the identification is achievable from merely positive examples. So far, grammar inference has been mainly successful in inferring regular languages. Researchers have developed various algorithms (e.g., RPNI - Regular Positive and Negative Inference algorithm [9]) which can learn regular languages from positive and negative examples. Learning context-free grammars (CFG) is more difficult than learning regular grammars. Using *representative* positive examples (that is, positive examples which put into effect every production rule in the grammar) along with negative examples did not result in the same level of success as with regular grammar inference. To be more successful, researchers used additional knowledge to assist in the induction process. They used a set of skeleton derivation trees (unlabeled derivation trees) [10], partially structured sentences [11], evolutionary process with initial population of heuristically build grammars, specialized genetic operations in evolutionary process, etc. Despite the fact that many researchers have looked into the problem of CFG induction, it remains a challenge in grammatical inference. One of the latest successful attempts to infer DSL grammars is the GIE tool, which will be described and modified for the web service presented in this paper.

### 2.2.2 GIE tool

GIE is an interactive tool for grammatical inference of context-free grammars. Its general idea is to search all possible labelled derivation trees for the selected example. This process consists of two steps. The first is to build an unlabeled tree and the second is to label it. When the tree is labeled we can obtain grammar from it (Figure 1).

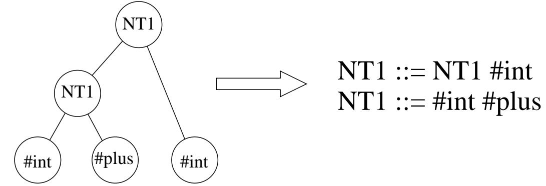


Figure 1: Extracted grammar from labeled tree

From obtained grammar we can then build a parser and test it on other examples. If the parser recognizes all positive examples as true, and rejects all negative examples, the grammar inference process was successful. However, to fully accomplish the inference process, we also need the following:

- Structurally complete positive examples; it is not possible to infer a production in a grammar if the positive example does not evidence it.
- Pool of negative examples needs to be complete; it is possible that inferred grammar is too general for specific application domain.

Process of inferring a grammar with the GIE is interactive and adaptive (Figure 2). To use the tool a user needs to follow five steps.

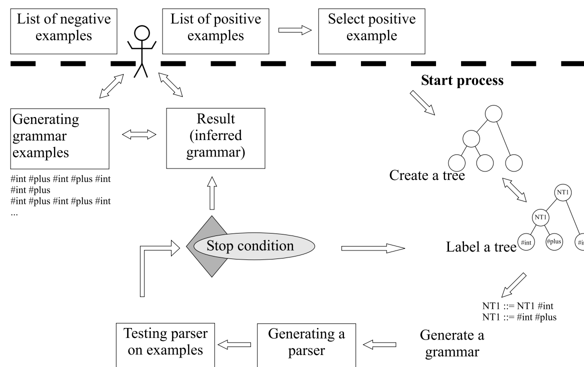


Figure 2: Interactive adaptive process of inferring a grammar

The first step is to write a list of positive and negative examples, collect regular expressions for basic symbols of searched grammar (number, word,

etc.), select the most complex positive example, and determine a stop criteria (usually this is a case when the parser recognizes all positive examples and reject all negative examples). The second step is to begin the process of inferring a grammar. The process stops at the first grammar that satisfies the stop criteria. There is an additional option to find all grammars that satisfy the stop criteria. The third step is to validate the grammar. This step needs some domain expert knowledge. To validate the grammar the user must be familiar with the grammar form. To simplify this step the GIE may generate examples from inferred grammar. This enables experts to check all generated examples and put wrong ones in a list of negative examples. The final step is to rerun the process until the searched grammar is found. The main constraint of GIE is time consumption which is exponential with structurally complete example of the searched language. GIE is working well on relatively small languages like robot, nesting blocks, DESK, etc. [2].

# 3 Web Service

During studying various approaches for the development of domain specific languages, we discovered many unsolved problems. Even though compiler construction is a well understood discipline, language design and implementation remains a demanding and unpopular task. The latter is particularly important because many of domain specific languages are being used in areas other than computer science [6]. Tools for automatic generation of parsers and compilers such as YACC, Lex, ASF+SDF, and LISA can greatly simplify and speedup the entire process of language development. Despite the abundance of such tools the general opinion of domain specific language developers remains the same – negative. To overcome these drawbacks we developed an intelligent web service for automatic generation of compilers. At the core of the web service lies LISA and GIE system. LISA itself is a very complex tool which, in addition to compiler generation, offers many advanced features not suitable for non–expert users. The developed web service brings these features to beginners through its intuitive user interface. During the design phase of web service the following was taken into consideration:

- A more simplified usage of compiler generator than with classical compiler generator tools (there is no need for installation and deeper knowledge of the tool).
- The development of programming language specification should be more straightforward.
- The need for a central repository for storing programming language specifications and an interface for searching those specfications.
- The possibility of developing a parser from sets of positive and negative program examples.

There are of course certain limitations to the web service, such as:

- The majority of visualization tools are not available through the web service.
- Transferring large amounts of data via network connections.

The main functionality of the web service is to generate a compiler from user defined formal programming language specifications. The generated compiler includes lexical analyzer (scanner), parser, and evaluator. Special consideration has been put into the development of user interface. The idea of user interface was to guide the user through the entire development process of a programming language. The basic development process is very similar to that in LISA:

- Writing specifications from scratch or importing them remotely (Figure 3).
- Generating scanner, parser, evaluator, and compiler. The result of this step are source Java files and their compiled versions.
- Error reporting at all levels.
- Downloading of generated components to user.
- Testing the generated compiler (compiling programs written in a new language).
- Executing the programs and inspecting the results.

To alleviate the task of language development even further we provided a central repository for storing specifications and then reusing them in other

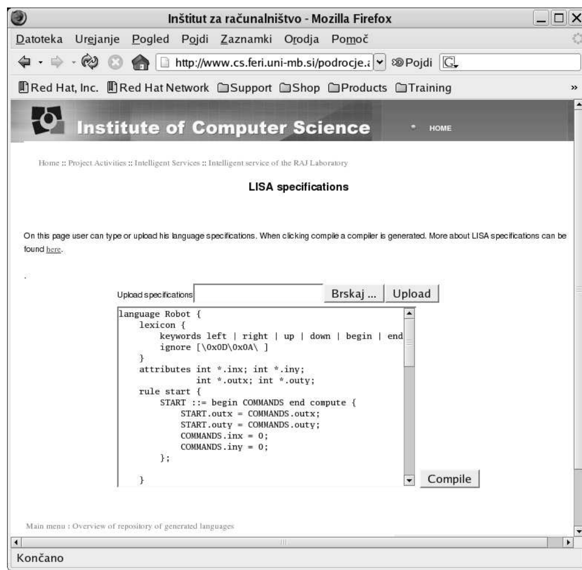


Figure 3: Web Service input form

languages. The language developer can inspect repository for specifications before he/she starts writing language specifications from scratch. Before storing them to the repository the specifications need to be thoroughly documented and classified into appropriate domain. Domains can be freely maintained by web service users. Each domain is represented with the following attributes: domain name, detailed description and purpose of languages in this domain, and representative words of domain description (keywords). After choosing the appropriate domain, specifications can be added to the repository. Before storing them into a specific domain specifications must be well documented. The documentation includes detailed description, keywords and author information. Specifications can also include example programs. Existing programming languages stored in a repository can be reused due to the fact that the LISA subsystem and the web service itself use multiple attribute grammar inheritance [7].

The repository of specfications can be browsed to find the appropriate solution in one of its domains. Specifications are divided into different domains, depending on the nature of programming language. Another way to search for specifications is to use the built–in web service search engine. Search can be performed using the following criteria:

- Domain.
- Keywords.
- Description (developer's comments) of programming language.
- Username (developer's user name).

The search can include all search terms or just specific ones.

The described functionalities greatly simplify the use of a compiler generator but still do not offer sufficient support to unexperienced users. Many times we have example programs which we want to test for syntactic correctness. For this purpose we need a parser. A standard procedure for developing a programming language is to build a scanner/parser from specifications. The alternative way is to build a scanner/parser from a set of example programs (positive – $P^+$ and negative – $P^-$). Therefore, we developed an interface which is, with the help of GIE, capable of guiding a user through the process of generating a parser from positive and negative example programs. Generating a parser from examples consists of the following steps:

- Importing negative and positive program examples set.
- Generating a parser from input.
  - Chooosing parameters (suggestions from the web service).
  - Reviewing results and further tuning.
- Downloading the generated components.
- Testing the generated components.

Web service usage is depicted in figure 4.

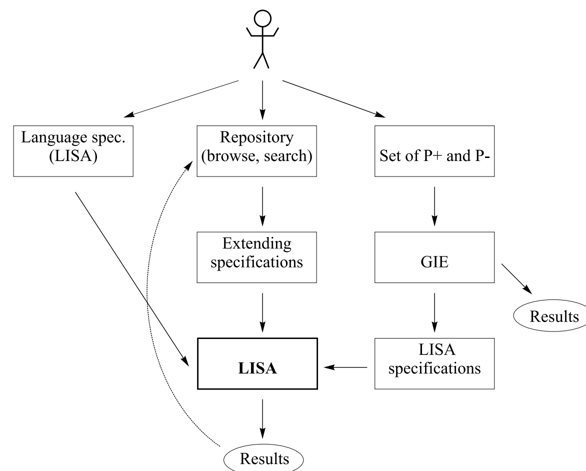


Figure 4: Web Service Usage

## 4 Conclusion and future work

We presented an intelligent web service for the development of programming languages. The web service is based upon LISA and GIE tools for automatic generation of compilers. To overcome the limitations and drawbacks of those tools we designed an intelligent user interface which greatly simplifies the entire developmental process of programming languages. The web service has been designed to be sufficiently useful for beginners as well as experts. Special interest has been put into integrating the GIE component for generating parsers from example programs with our web service. The GIE is undoubtedly the most intelligent part of the web service.

We noticed several challenges for future work on our web service. The main feature of the web service still required is robustness. Improvement of user interface is also among the things to be done. In particular we wish to improve the intelligence of the interface as it is the key to user friendliness. The interface will be extended with a component which will enable programming language development in a visual manner. A forum for developers is also planned.

Despite that fact that many issues still need to be addressed we believe our web service became an important contribution to the compiler generating tools.

## 5 Acknowledgements

We would like to thank Mitja Lenič for his tremendous contribution to the LISA component of the web service.

## References


[1] M. Črepinšek, M. Mernik, F. Javed, B. Bryant, and A. Sprague. Extracting grammar from programs: Evolutionary approach. *ACM Sigplan*, 40:39–46, 2005.

[2] M. Črepinšek, M. Mernik, and V. Žumer. Extracting grammar from programs: Brute force approach. *ACM Sigplan Notices*, 40:29–38, 2005.

[3] E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[4] L. Lee. Learning of context-free languages: a survey of the literature. Technical Report TR-12-96, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1996.

[5] J. R. Levine, T. Mason, and D. Brown. *Lex and Yacc*. Nutshell Handbook. O'Reilly and Associates, Sebastopol, California, U.S.A., 2nd edition, 1992.

[6] M. Mernik, M. Črepinšek, T. Kosar, D. Rebernak, and V. Žumer. Grammar-based systems: Definition and examples. *Informatica*, 28(3):245–254, 2004.

[7] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Multiple Attribute Grammar Inheritance. *Informatica*, 24(3):319–328, September 2000.

[8] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. LISA: An Interactive Environment for Programming Language Development. In Nigel Horspool, editor, *11th International Conference on Compiler Construction*, volume 2304, pages 1–4. Lecture Notes in Computer Science, Springer-Verlag, 2002.

[9] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In N. Pérez de la Blanca, A. Sanfeliu, and E. Vidal, editors, *Pattern Recognition and Image Analysis*, volume 1 of *Series in Machine Perception and Artificial Intelligence*, pages 49–61. World Scientific, Singapore, 1992.

[10] Y. Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60, March 1992.

[11] Y. Sakakibara and H. Muramatsu. Learning context-free grammars from partially structured examples. In *Grammatical Inference: Algorithms and Applications, 5th International Colloquium, ICGI 2000, Lisbon, Portugal, September 11 - 13, 2000; Proceedings*, volume 1891 of *Lecture Notes in Artificial Intelligence*, pages 229–240. Springer, 2000.

[12] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF + SDF meta-environment: A component-based language development environment. *Lecture Notes in Computer Science*, 2027:365–370, 2001.