UDC 004.4, DOI: 10.2298/CSIS1002291P

Annotation Based Parser Generator*

Jaroslav Porubän[1], Michal Forgáč[1], Miroslav Sabo[1], and Marek Běhálek[2]

[1] Department of Computers and Informatics, Technical University of Košice,
Letná 9, 042 00 Košice, Slovak Republic
{Jaroslav.Poruban, Michal.Forgac, Miroslav.Sabo}@tuke.sk
[2] Department of Computer Science, FEI VŠB Technical University of Ostrava,
17. listopadu 15, 708 33 Ostrava-Poruba, Czech Republic
marek.behalek@vsb.cz

Abstract. The paper presents innovative parser construction method and parser generator prototype which generates a computer language parser directly from a set of annotated classes in contrast to standard parser generators which specify concrete syntax of a computer language using BNF notation. A language with textual concrete syntax is defined upon the abstract syntax definition extended with annotations in the presented approach. Annotations define instances of concrete syntax patterns in a language. Abstract syntax of a language is inevitable input of the parser generator as well as language's concrete syntax pattern definitions. The process of parser implementation is presented on the concrete computer language – the Simple Arithmetic Language. The paper summarizes results of the studies of implemented parser generator and describes its role in the university courses.

Keywords: parser generator; annotated model; abstract syntax; model to grammar transformation.

1. Introduction

Computer languages are crucial tools in the development of software systems. By using computer languages we define the structure of a system and its behavior. Today's common industry practice is to create a software system as a composition of software artifacts written in more than one computer language. Developers use different languages and paradigms throughout the development of a software system according to a nature of concrete subproblem and their preferences. Besides the general-purpose programming languages (e. g. Java, C#) the domain-specific languages (DSL) [1][2] have become popular in the last decade. Nowadays, DSLs have their stable position in the development of software systems in many different

* This work is supported by APVV Grant No. SK-CZ-0095-07 – Cooperation in Design and Implementation of Language Systems and by VEGA Grant No. 1/4073/07 – Aspect-oriented Evolution of Complex Software System.

forms. Concerning abstraction level, it is possible to program closer to a domain. Furthermore DSLs enables explicit separation of knowledge in the system in natural structured form of domain. The growth of their popularity is probably connected with the growth of XML technology and using of standardized industry XML document parsers as a preferable option to the construction of language specific processors. A developer with minimal knowledge about language parsing is able to create a DSL with XML compliant concrete syntax using tools like JAXB [3].

Computer languages come in many flavors – as well known GPLs, DSLs, but also as APIs, ontologies [4], and even others. The one of the today's hottest research topics in the field of computer language development is the tooling support. In the paper we concentrate on the parser generators for DSLs. Even though the research in the field of computer languages has the long history and parser generators for a textual language processing like YACC [5], Bison [6], JavaCC [7] and ANTLR [8] have their stable position in the computer language development the task of developing a computer language is still an expert task. Cook et al. [9] conclude that implementing a textual DSL by implementing its grammar can be a difficult and error-prone task, requiring significant expertise in language design and the use of a parser generator. Similarly, Mernik et al. [1] argue that DSL development is hard, requiring both domain knowledge and language development expertise. We present the novel method of a computer language design and implementation in the paper – *abstract syntax driven parser generation*.

The rest of the paper has the following structure: In the section 2 we present main ideas behind our approach to a computer language development. The section 3 explains the method on example of simple but extensible arithmetic language. Section 4 describes the parser generator prototype –*YAJCo*. Section 5 summarizes the results of our experiments with *YAJCo* parser generator. Section 6 compares our work with the state of the art in the field of parser generators. The last section 7 concludes the paper and outlines the possibilities for further research in the field of parser generators and computer language development in general.

2. Abstract Syntax Directed Language Definition

This section sketches the main ideas behind the innovative approach to the definition of a concrete syntax for a computer language with textual notation. Contrary to traditional methods of parser generation (e. g. YACC, JavaCC), we focus on the definition of abstract syntax rather than giving an excessive concentration on concrete syntax (see Fig. 1). In our approach the abstract syntax of a language is formally defined using standard classes well known from object-oriented programming and metamodels. Kleppe argues for concentrating on abstract syntax and metamodels when we define a computer language in [10].

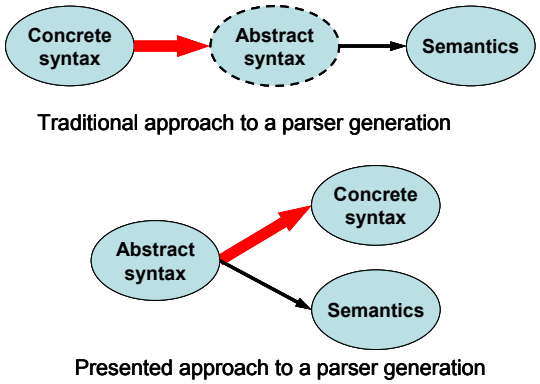

Fig. 1. Comparing traditional and presented approaches to a computer language parser generation

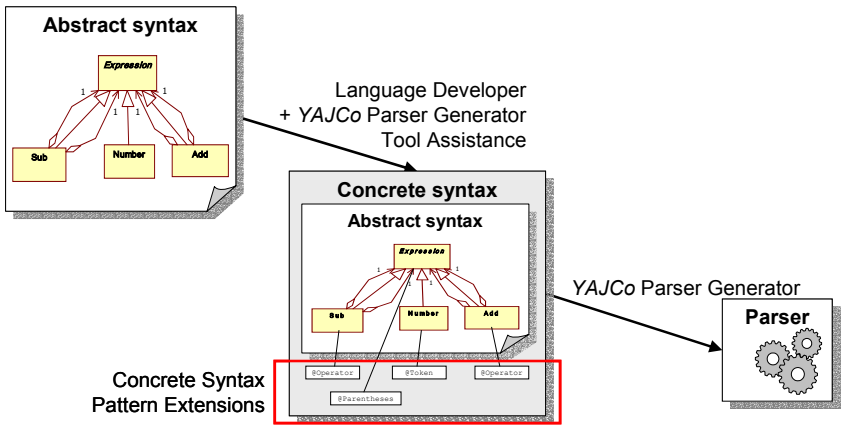

Fig. 2. Generating Language Parser using YAJCo's parser generation approach

In our approach the language implementation begins with the concept formalization in the form of abstract syntax. Language concepts are defined as classes and relationships between them. Upon such defined abstract syntax a developer defines both the concrete syntax through a set of source code annotations and the language semantics through the object methods. Annotations (called also attributes [11] are structured way of additional knowledge incorporated directly into the source code. During the phase of concrete syntax definition the parser generator assists a developer with suggestions and hints for making the concrete syntax unambiguous. Fig. 2 shows the whole process of parser implementation using the described approach. If the concrete syntax is unambiguously defined then parser generator automatically generates the parser from annotated classes.

It is quite common to have multiple notations for one language. RELAX NG [12] is an example of such a language with two different notations – XML

concrete syntax and compact concrete syntax. By using our approach different notations of the same language can share both abstract syntax and semantics. In some cases the evolution of concrete syntax does not require the modification of abstract syntax and semantics at all. This means that other notations of the same language are not affected by this type of language evolution. For instance, Fig. 3 presents the language with four different notations sharing the same abstract syntax and semantics. These notations (concrete syntaxes) are textual notation, XML notation, in-memory object notation and graphical notation. Concrete notations are interchangeable and developer selects among them according to his preferences.

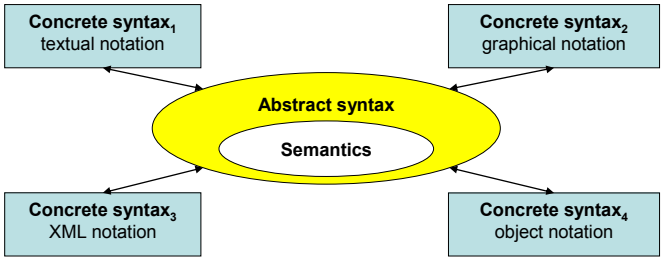

Fig. 3. Computer language with multiple notations and shared abstract syntax and semantics

3. SAL Example

This section presents our approach to a computer language definition using annotated classes on the example of Simple Arithmetic Language (SAL). This language expresses the arithmetic expressions with basic arithmetic binary operations of addition and multiplication, and unary operation of arithmetic negation. The expressions also contain integer numbers. The abstract syntax of SAL can be formally defined using BNF as follows.

$e, e_1, e_2 \in$ Expression, $n \in$ Number
e ::= Number n | UnaryMinus e | Add e_1 e_2 | Mul e_1 e_2

Variables e, e_1, e_2 are metavariables from the **Expression** syntactical domain and n is the metavariable from the **Number** syntactical domain. The infix form of arithmetic operation is intentionally omitted to avoid the confusion with concrete syntax. Prefix names in productions (e. g. UnaryMinus, Mul) are used just to uniquely name the productions for semantic equations.

The semantics of SAL is formally defined using *Eval* function which maps a value from syntactic domain **Expression** to a value from semantic domain **Z** (integers) and *Value* function which maps a value from syntactic domain **Number** to a value from semantic domain **Z**.

Eval : Expression → Z
Value : **Number → Z**
The semantic function *Eval* is defined by the following equations.
Eval ⟦ Number *n* ⟧ = *Value* ⟦ *n* ⟧
Eval ⟦ UnaryMinus *e* ⟧ = – *Eval* ⟦ *e* ⟧
Eval ⟦ Add e_1 e_2 ⟧ = *Eval* ⟦ e_1 ⟧ + *Eval* ⟦ e_2 ⟧
Eval ⟦ Mul e_1 e_2 ⟧ = *Eval* ⟦ e_1 ⟧ * *Eval* ⟦ e_2 ⟧

Certainly we can find many different notations for SAL. For example, we can write down a sentence from SAL in the following notation using standard symbols and the operator infix form.

```
1 + 2 * 7
```

In the Fig. 4, abstract syntax tree of the sentence above is depicted.

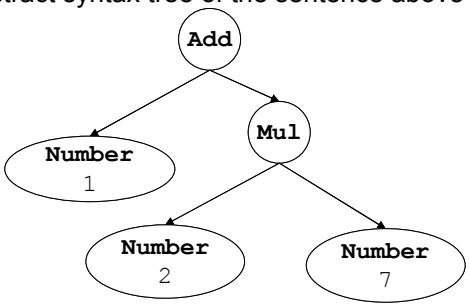

Fig. 4. The abstract syntax tree of the expression `1 + 2 * 7`

From the Fig. 4 it is apparent that depicted abstract syntax tree contains typed nodes corresponding to language concepts. The node types are Add, Mul and Number. Add node represents the binary operation of addition. It always has two child nodes respecting the nature of the binary operation of addition. The Mul node represents multiplicative operation and the leaf node Number represents an integer number. The Number node is attributed with the notation of a number.

Unlike traditional approach, language definition will not start with the definition of SAL's concrete syntax written in BNF. According to our approach, the object classes representing syntactic domains (language concept) are created at first. These classes define the abstract syntax of the language and also the semantics of the language as stated in the previous formal definition of the SAL. The concrete syntax will be specified later using source code annotations, expressing the concrete syntax patterns and their correspondence to the abstract syntax concepts.

The main concept of the SAL is the `Expression`. It is pretty straightforward because SAL is the language of expressions. On the other side it is an abstract concept and it does not have concrete representation. From the semantic point of view every expression can be evaluated to a single integer value. This fact is denoted by semantic function `eval` of `Expression` class.

```
abstract class Expression {
  //Semantic function – OOP method
  abstract int eval();
}
```

The `Expression` class is declared to be abstract because it only defines the abstract concept of an expression from SAL and does not represent any abstract syntax graph node. Next, the different types of expressions can be incorporated into the SAL. The simplest form of an expression is a number expression. Number has its notation and the value. Firstly we will focus is on its value. The notation will be defined later during the definition of the concrete syntax. It needs to be expressed that number is a simple expression as well. This is done using "is-a" relationship, denoted with **extends** keyword in Java. Corresponding semantic equations are denoted in the comments above the methods. The code snippet below shows the class `Number` for integer numbers.

```
class Number extends Expression {
  int value;

  //Eval [| Number n |] = Value [| n |]
  int eval() {
    return value;
  }
}
```

The unary operation of negation is defined in the following snippet of the `UnaryMinus` class.

```
class UnaryMinus extends Expression {
  Expression expression;

  //Eval [| UnaryMinus e |] = – Eval [| e |]
  int eval() {
    return -expression.eval();
  }
}
```

Since the addition is a kind of arithmetic expression in SAL, the binary operation of addition is defined in the class `Add`. Relationship "is-a" is therefore used again.

```
class Add extends Expression {
  Expression expression1;
  Expression expression2;

  //Eval [| Add e1 e2 |] = Eval [| e1 |] + Eval [| e2 |]
  int eval() {
    return expression1.eval() + expression2.eval();
  }
}
```

Operation of multiplication is defined in the same style as binary operation `Add`.

The class diagram in the Fig. 5 shows the hierarchy of SAL classes. The abstract syntax of arithmetic expression language has already been defined as well as the semantic function *Eval* using the classic OOP notation. The next step in the development of SAL is to define the concrete syntax for the language. Concrete syntax will be used when expression (sentence) will be stored in the textual form.

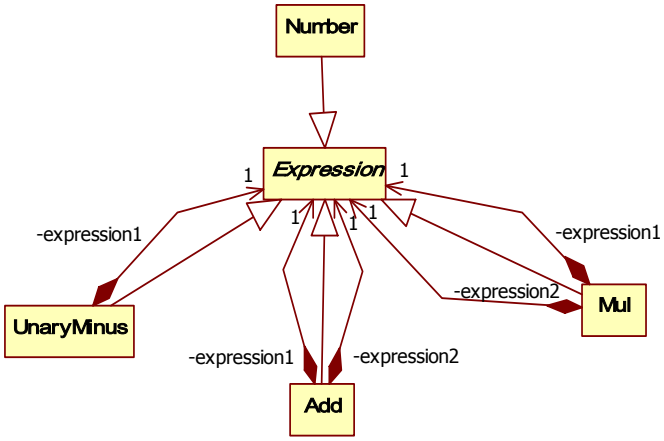

Fig. 5. Classes and their hierarchy in the simple arithmetic language (SAL)

The specification of concrete syntax requires some additional information about textual representation of the language concepts. In SAL it is:

- a number representation (notation),
- notation for operations,
  - symbols for the operations of addition, multiplication and negation,
  - the form of the notation, the priority and associativity of all operations.

The operations will be expressed in infix form using standard symbols `+` and `*`. Unary operation of negation will be in the prefix form denoted with the symbol `-`. The priority, associativity and symbols for the operations are listed in Table 1. The integer numbers are written using standard decimal notation with digits `0`, `1`, …, `9`.

Table 1. Priority and associativity of SAL operators

Operator	Priority	Associativity
+	1 (lowest)	left
*	2	left
-	3 (highest)	right

The class for integer numbers is augmented with concrete syntax source annotations in the following code snippet.

```
class Number extends Expression {
  int value;

  Number(@Token("VALUE") long value) {
    this.value = value;
  }

  int eval() {
    return value;
  }
}
```

The `@Token` annotation with `VALUE` attribute defines the name of a regular expression for the number notation. As seen on the snippet the class constructor is augmented with the concrete syntax pattern. The regular expression can be defined as follows.

```
@TokenDef(name = "VALUE", regexp = "[0-9]+")
```

The format of a regular expression depends on the syntax for definition of regular expressions. The annotation `@Token("VALUE")` can even be omitted because the name of token can be derived directly from the name of the parameter (`value` in this case). The domain class for binary operation of addition augmented with concrete syntax annotations is shown below.

```
class Add extends Expression {
  Expression expression1;
  Expression expression2;

  @Operator(
    associativity = Associativity.LEFT,
    priority = 1
  )
  Add(Expression expression1,
      @Before("+")
      Expression expression2) {
    this.expression1 = expression1;
    this.expression2 = expression2;
  }

  int eval() {
    return expression1.eval() + expression2.eval();
  }
}
```

Concrete syntax for the operation of addition is defined in the class constructor. Parameters of constructor define the rule of composition of the operation. In the constructor body it can be observed that addition is composed of two expressions in textual form. It is important to notice that after the first expression (and before the second expression at the same time) token + will follow.

Binary operation of multiplication is defined accordingly to the definition of addition. The domain class for unary operation of arithmetic negation is

augmented with concrete syntax annotations as shown in the code snippet below.

```
class UnaryMinus extends Expression {
  Expression expression;

  @Operator(priority = 3)
  UnaryMinus(
      @Before("-")
      Expression expression) {
    this.expression = expression;
  }

  int eval() {
    return -expression.eval();
  }
}
```

As seen in the constructor the operation is defined as unary prefix operation.

The last step in definition of the SAL's concrete syntax is the definition for parentheses. This can be achieved simply by using the annotation on abstract class for expressions as shown below.

```
@Parentheses(left = "(", right = ")")
  abstract class Expression {
    //...
}
```

Finally the concrete syntax for the language has been defined. The implemented *YAJCo* parser generator generates the language parser from annotated classes. The concrete syntax of SAL is automatically derived from these classes, their relationships and concrete syntax annotations. In the current implementation of the *YAJCo* it is the following LL(1) context-free grammar.

```
Expr1 ::= Expr2 {"+" Expr2}
Expr2 ::= Expr3 {"*" Expr3}
Expr3 ::= "-" Expr3 | Expr
Expr ::= Number | "(" Expr1 ")"
Number ::= [0-9]+
```

4. *YAJCo* Parser Generator

The main goal of the approach is not to create a new parsing technology based on context-free grammars theory. The main idea is to integrate existing technologies into the higher level abstraction in which the language developer does not have to concentrate on concrete parsing technology but on the

language itself describing the concepts and relationships between them with abstract syntax in mind. The main characteristics of the approach are:

- Orientation on abstract syntax and semantics of the language.
- Definition of the concrete syntax independent from a parsing technology.
- Automatic construction of abstract syntax tree from an input sentence.
- Automatic construction of references between concept instances.
- Error reporting in terms of language domain concepts.
- Separation of language concepts on implementation level (concept types).
- Tool support for language evolution (concept refactoring).

As a proof of concept the parser generator *YAJCo* (Yet Another Java Compiler cOmpiler) has been implemented. *YAJCo* generates language parser from annotated classes. It is implemented as a standard Java annotation processor which traverses through the source code of classes looking for cocnrete syntax pattern annotations. *YAJCo* discovers relations between classes. Two main relationships between classes used in the definition of an abstract syntax are:

- "is-a" relationship,
- "has-a" relationship.

Together with corresponding BNF productions they are depicted in the Fig. 6.

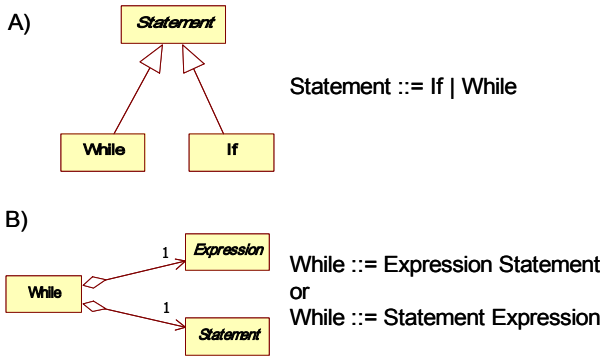

Fig. 6. Abstract syntax relationships: A) "is-a" relationship, B) "has-a" relationship

The "is-a" relationship is used also in the definition of concrete syntax, but the "has-a" relationship has following drawbacks when defining the concrete syntax:

- Multiple notations for a single concept.
- Lack of natural ordering for member variables defined in a class (except the order in a source code).
- Data type conversion between concrete and abstract syntax (e. g. dropping the quotes from the string literal).

All these drawbacks can be eliminated by using class constructor notation (or factory methods notation) for the definition of concrete syntax. This is the

main reason why we annotate constructors and their parameters instead of object fields as shown in the following example.

```
While(
  @Before({"while", "("})
  @After(")")
  Expression expr,
  Statement stmt) {…}
```

The previous example corresponds to the following BNF production of a concrete syntax.

```
While ::= 'while' '('Expression ')' Statement
```

To define a transformation from abstract to concrete syntax a set of concrete syntax annotation types has been created:

- Structural annotations – mark the concept as optional or set the minimum and maximum number of occurrences - `@Optional, @Range`
- Token annotations – specify binding of lexical units to abstract syntax concepts - `@Before, @After, @Token, @Separator`
- Language pattern annotations – identify common computer language patterns
  - Operators: `@Operator, @Parentheses`
  - Identifiers and references: `@Identifier, @References`
- Parser configuration annotations - `@Parser, @TokenDef, @Skip`

Following print statement example presents the usage of some of the annotations mentioned above.

```
class Print extends Statement {
  @Before("print")
  @After(";")
  Print(
    @Separator(",")
    @Range(minOccurs = 1)
    Expression[] expressions) { ... }
...
}
```

The corresponding `print` language concept has the following notation.

print $expr_1$, ... , $expr_n$;

The next example presents annotated C language `if` statement.

```
class If extends Statement {
  If(
    @Before({"if", "("})
    @After(")")
    Expression expression,
    Statement trueStatement,
    @Optional
    @Before("else")
    Statement falseStatement) {...}
```

```
...
}
```

Currently the JavaCC parser generator is used as the underlying parsing technology. As an output *YAJCo* generates JavaCC grammar file augmented with actions for constructing abstract syntax tree. Since the annotations are independent of concrete parsing technology the output can also be generated for other top-down or bottom-up parser generators (e. g. ANTLR [8]).

Finally the parser for SAL with tokens and blank characters is defined using `@Parser` annotation as shown in the code snippet.

```
@Parser(
  className = "parser.expr.ExpressionParser",
  rootNode = "Expression",
  tokens = {
    @TokenDef(name = "VALUE", regexp = "[0-9]+")
  },
  skips = {
    @Skip(" "),
    @Skip("\t"),
    @Skip("\n")
  }
)
```

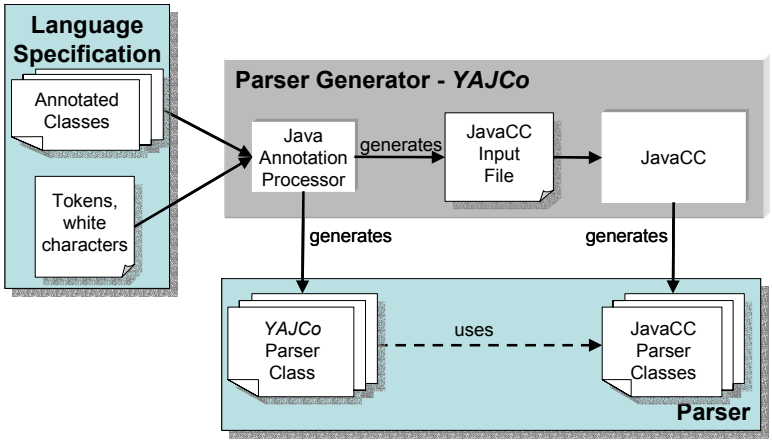

Fig. 7. Generating parser using YAJCo parser generator – YAJCo architecture overview

Processing of annotated classes with developed parser generator *YAJCo* is depicted in the Fig 7. After the generation of parser is complete it can be simply embedded in any existing Java application. The following code snippet is an example of embedding generated source code parser for SAL.

```
String expr = "1 + 2 * 7";
Expression expression = new
          ExpressionParser().parse(expr);
long result = expression.eval();
```

5. Experiments

To explore the full potential of the implemented approach and *YAJCo* parser generator we have implemented seven computer languages, each of them having a different character:

- SAL – Simple Arithmetic Language,
- AL – Arithmetic Language,
- SIL – Structured Imperative Language,
- PIL – Procedural Imperative Language,
- GUIIL – Graphical User Interface Interaction Language,
- SML – State Machine Language,
- LAD – Language of Annotation Designator.

SAL [17] and AL languages are simple computer languages for expressing the arithmetic expressions. AL has been created incrementally from the SAL in a few evolutionary steps. In every step some new constructs have been incorporated into the language. SIL and PIL languages are the representatives of general-purpose programming languages. PIL is procedural Pascal-like language. These languages are greatly inspired by traditional university compiler course languages. On the other hand the last three languages GUIIL, SML and LAD are DSL languages oriented to concrete domains. GUIIL is the language which describes the recipes for graphical user interface task automation. SML is classic DSL for state machines description [9]. LAD is DSL language for expressing the annotation constraints. All mentioned languages were successfully implemented using *YAJCo* parser generator. During the implementation of these computer languages we have also defined some metrics to measure the following implementation characteristics:

- number of language concepts (defined by types - classes, interfaces, enumerations),
- number of annotations in the implementation of language concepts categorized by annotation types,
- comparison of annotated and unannotated language concepts,
- characteristics of generated source code (number of source lines of code, number of characters).

The results of experiments are summarized in Table 2 According to our measurements the most complex language is PIL. This language contains the largest number of language concepts. From the point of view of the number of language concepts the simplest languages are SML and GUIIL. According to the results the most common language concept representation is a concrete class. Interfaces and abstract classes are interchangeable by the choice of language developer. The most common concrete syntax annotation used in experimental languages is `@Before`. This is a reasonable outcome since the annotation specifies the lexical symbol preceding a concept. It is natural to specify the concept with leading keyword (e.g. if, while, procedure). The interesting fact is that approximately 25% of language concept types contain no annotation. It is the fulfillment of the one of our aims - to minimize the

number of used annotations. The results also show that the SML language is considerably verbose. The average number of concrete syntax annotations per concept type in SML is 2.5. The following part from the SML sentence presents the level of verbosity of the SML language.

Table 2. Results from the implementation of experimental languages using YAJCo parser generator

Types	**SAL**	**AL**	**SIL**	**PIL**	**GUIIL**	**SML**	**LAD**
Concrete class	6	11	30	42	4	6	26
Abstract class	1	3	2	3	1		7
Interface				2			
Enumeration			1				1
Total	7	14	33	47	5	6	34
Annotation	**SAL**	**AL**	**SIL**	**PIL**	**GUIIL**	**SML**	**LAD**
After		1	9	13	1	1	13
Before	5	10	28	37	2	8	23
Operator	5	10	17	27			8
Optional			1	1	1	1	3
Parentheses	1	1	1	1			2
Range			3			2	2
Separator			3	2	1		5
Token			2			3	2
Total	11	22	64	81	5	15	58
Category	**SAL**	**AL**	**SIL**	**PIL**	**GUIIL**	**SML**	**LAD**
Number of annotated types	6	11	25	39	2	4	23
Number of types without annotation	1	3	8	8	3	2	10
Average number of annotations per type	1.57	1.57	1.94	1.72	1.00	2.50	1.71
Ratio of unannotated types to all types	0.14	0.21	0.24	0.17	0.60	0.33	0.29
Characteristics	**SAL**	**AL**	**SIL**	**PIL**	**GUIIL**	**SML**	**LAD**
Number of lexical units	8	16	37	40	7	11	37
Number of BNF rules	5	7	24	27	5	6	29
Number of source lines of code generated by *YAJCo*	128	187	570	655	124	168	693
Number of characters generated by *YAJCo*	2458	3775	15912	17554	2854	4245	19559
Number of source lines of code generated by	1487	1603	2567	2580	1516	1843	3849

JavaCC							
Number of characters generated by JavaCC	42103	45370	78450	78687	43869	52950	114002

```
transition from Ready to Running when water_high
```

The average usage of concrete syntax annotation per one concept type in all languages is less than 2. The main goal of the metrics definition was the measurement of a language complexity based on abstract syntax since abstract syntax directly defines concepts from a domain.

The successful implementation of experimental languages proves the viability of *YAJCo* parser generator. That was the main reason why we decided to incorporate the tool in the university master course concerning DSL implementation and model driven software development. More than 30 students have successfully used the *YAJCo* parser generator as a part of their projects.

6. Related Works

Currently there are a lot of parser generators for various programming languages [5][6]. Classic parser generators like JavaCC [7] generate the parser as a single huge class ignoring the concept of composition of language concepts and concentrating on the concrete syntax of a language. These tools are still greatly inspired by procedural nature of YACC-like tools. The concrete syntax is specified in DSL of parser generator. It is usually a language for writing the context free grammar enriched with constructs for language semantics definition. During language development the developer is often dealing with the type of parsing algorithm which is supported by selected parser generator (e. g. LL, LR, LALR) and his decisions are forced by the type of grammar supported by the tool. Even JJTree, a tool provided by JavaCC for generating the abstract syntax tree from the textual representation, is still driven by the point of view of concrete syntax grammar rather than abstract syntax language concept. Consequently, changes made to grammar must be also reflected in the representation of abstract syntax nodes in programming languages. The semi-automatic refactoring of generator's DSL is still missing.

On the other side, there is a notable growth in the field of language workbenches [13] on the market. MDSD [14] tools like Microsoft Visual Studio DSL Tools (software factories representative [15]) are being incorporated into the programming IDEs. The primary orientation of these tools is graphical notation of computer languages. However, the special support for textual language notation is not provided.

Authors in [16] propose another approach to mapping from abstract syntax to concrete and back. Their solution is based on complex language rather than concrete syntax patterns.

7. Conclusion

In the paper we have presented solution for generating parsers for textual languages. The language itself is specified by a set of annotated classes. Annotations extend the classes with additional information required for specification of concrete syntax, for example keywords and operator notations. The developer can start with the definition of abstract syntax and continue with creation of language in incremental way using the standard refactoring tools. In proposed solution there is only one form of definition of abstract syntax graph nodes – by the classes. The grammar is derived directly from the source code of annotated domain classes. Even the examples are written in object-oriented programming language Java our solution is not strictly connected to Java language and can be easily ported to any other object-oriented language supporting the attribute-oriented programming. We believe that our solution can simplify the development of textual software languages.

References

1. Mernik, M., Heering, J., Sloane, A. M.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys, Vol. 37, No. 4, 316–344. (2005)
2. Pereira, M. J. V., Mernik, M., da Cruz, D., Henriques, P. R.: Program Comprehension for Domain-Specific Languages. ComSIS, Vol. 5, No. 2. (2008)
3. Ort, E., Mehta, B.: Java Architecture for XML Binding. Sun Microsystems, [Online]. Available: http://java.sun.com/developer/technicalArticles/WebServices/jaxb (current November 2009)
4. Návrat, P., Bieliková, M., Chudá, D., Rozinajová, V.: Intelligent Information Processing in Semantically Enriched Web. Lecture Notes in Computer Science, Vol. 5722/2009, 331-340. (2009)
5. Johnson, S. C.: YACC: Yet Another Compiler-Compiler. Unix Programmer's Manual Volume 2b. (1979)
6. Donnelly, C., Stallman, R.: Bison: The Yacc-compatible Parser Generator. (2006).
7. Java Compiler Compiler – The Java Parser Generator, (2009). [Online]. Available: https://javacc.dev.java.net (current November 2009)
8. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages, Pragmatic Bookshelf, 376 pp. (2007)
9. Cook, S., Jones, G., Kent, S., Wills, A. C.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional, 576 pp. (2007)
10. Kleppe, A. G.: A Language Description is More than a Metamodel. In: Fourth International Workshop on Software Language Engineering, 1 Oct 2007, Nashville, USA.
11. Cepa, V.: Attribute Enabled Software Development, VDM Verlag, 216 p. (2007)
12. van der Vlist, E.: Relax NG. O'Reilly Media, 304 pp. (2003)
13. Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? (2005) [Online]. (current November 2009) Available: http://www.martinfowler.com/articles/languageWorkbench.html
14. Stahl, T., Voelter, M.:Model-Driven Software Development: Technology, Engineering, Management. Wiley, 444 p. (2006)

15. Greenfield, J., Short, K., Cook, S., Kent, S., Crupi, J.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, 500 p. (2004)
16. Muller, P. A., Fondement, F., Fleurey, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J. M.: Model-Driven Analysis and Synthesis of Textual Concrete Syntax. Journal on Software and Systems Modeling (SoSyM), Volume 7 (4), Springer, 423-441. (2008)
17. Kollár, J., Václavík, P., Wassermann, Ľ.: Data driven Executable Language Model. In Proceedings of the International Multiconference on Computer Science and Information Technology, Mragowo, Poland, IEEE Computer Society Press, 667-675. (2009), ISSN 1896-7094.

Jaroslav Porubän is Associate professor at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his MSc. in Computer Science in 2000 and his PhD. in Computer Science in 2004. Since 2003 he is the member of the Department of Computers and Informatics at Technical University of Košice. He was involved in the research of profiling tools for process functional programming language. Currently the main subject of his research is the computer language engineering concentrating on design and implementation of domain-specific languages and computer language composition and evolution.

Michal Forgáč is Assistant professor at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his MSc. in 2006 and his PhD. in Computer Science in 2009. Since 2009 he is the member of the Department of Computers and Informatics at Technical University of Košice. His scientific research is focused on the software evolution, software language engineering and adaptation of complex software systems.

Miroslav Sabo is doctoral student at Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his MSc. in Computer Science in 2008. The subject of his research is the utilization of generative methods in development and evolution of software systems in permanently changing environment.

Marek Běhálek is Assistant professor at Department of Computer Science, FEI VŠB Technical University of Ostrava, Czech Republic. He received his MSc. in 2002. Since 2004 he is the member of the Department of Computer Science at Technical University of Ostrava. His scientific research is focused on programming languages, their evolution and application. Currently he is developing a tool for modeling of embedded systems based on functional programming paradigm.

Received: November 16, 2009; Accepted: December 25, 2009.