Modernizing Parsing Tools

Parsing and Analysis with Object-Oriented Programming

Steven O'Hara
Rocky Slavin
steven.ohara@utsa.edu
rocky.slavin@utsa.edu
Department of Computer Science
University of Texas at San Antonio
United States

Abstract

Software Engineering tools today are hampered by weaknesses in parsing and analysis tools. For example, there are no standard repositories of grammars for the most popular programming languages. If an organization has software written in Python, Java, Bash, SQL, HTML, CSS, JavaScript and so on, there is no readily available mechanism to parse and analyze all of the software in a unified manner. This paper describes a collection of tools for parsing and analyzing many different languages, including legacy languages like COBOL and Fortran. The primary goal is scalability; dealing with a single programming language and a limited number of programs is far simpler than dealing with millions of lines of code written in many different languages.

CCS Concepts • **Software and its engineering → Software notations and tools**.

Keywords software analysis, parsers, modernization

ACM Reference Format:
Steven O'Hara and Rocky Slavin. 2019. Modernizing Parsing Tools: Parsing and Analysis with Object-Oriented Programming. In *Proceedings of the 8th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3315568.3329967

1 Introduction

Parsing tools like yacc[8] and ANTLR[16] have been widely used but they depend on grammars which have not been standardized and published for general purpose use. There is an expectation that each organization will create or adapt grammars for the languages they use. Many are available for various languages such as for Python[1] and COBOL[2], but they use different style grammars and generate language-specific structures.

The purpose of this research is to identify techniques such that all programming languages can be parsed using a single parser, and that the resulting parse trees are written in a consistent manner. In addition to the parser, we provide a collection of tools for more robust analysis. For example, all identifier definitions and references are stored in a language independent manner. Likewise, most terminal nodes (numbers, strings, punctuation, keywords, etc) are processed by shared libraries.

The core parser described in this paper, first introduced in [13], is compared to popular parsers based on LALR and LL(*) style parsers. Since our system uses recursive descent to parse, it is similar to LL(*) parsers such as ANTLR. There are two major differences between our system and traditional parsers. First, we use Reflection to infer the grammar from Java classes and fields, rather than a text file. Second, rather than generate parse output as a text file, we generate instances of Java classes, using the identical classes that were used for parsing.

Scalability is the focus of our approach. We are designing a collection of tools to enable the consistent analysis of major repositories of software. All of the tools we discuss are designed to handle many millions of lines of code, written in a multitude of languages. Tools like debuggers, tracers, dashboards, etc. are all described.

The majority of our work will be made open-source. In particular, all the Program Grammars will be modifiable by the community, with moderator supervision. The central parser itself has been patented[15].

The long range goal in this effort is exemplified by [12], which discusses the benefits that are available when grammars and analysis tools become standardized to the point that software can be reliably modified dynamically without human intervention. For example, migration of language

SOAP '19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6720-2/19/06…$15.00
https://doi.org/10.1145/3315568.3329967

[1] https://docs.python.org/3/reference/grammar.html
[2] https://open-cobol.sourceforge.io/guides/grammar.pdf

versions due to deprecated features should be completely automated.

2 Background and Related Work

There are hundreds of billions of lines of code in use today[17], many of which are in antiquated languages such as COBOL, RPG and Natural. The original developers are unlikely to still be available to maintain these systems and it is difficult to find or train legacy programmers. Likewise, there is an on-going proliferation of languages, both general purpose and domain-specific[10]. The technologies presented here are equally applicable to modern programming languages, which continue to evolve as well.

To get enterprise-wide analyses in such environments requires spanning both legacy and modern systems. The following describes the state of the art for addressing this issue.

2.1 Parsing Tools

The first step in most software analysis projects is to parse the source code base. Most current parsers use context-free grammars, which are less expressive than ours, which supports context-sensitive grammars.

The two most popular approaches to parsers are Look-Ahead Left-to-Right (LALR)[5] such as yacc, and LL(*)[3] such as ANTLR. Our approach is more closely related to the LL(*) style since both are essentially recursive descent parsers. Unlike LALR, we do not do any lexical token look-ahead.

One of the criticisms of LL(*) grammars, including ours, is the inability to handle left-recursive grammars, such as "`expr ::= expr '+' expr | expr '-' expr;`". However, it is our assessment that this normally happens while processing expressions, so we provide a specialized parser that allows expressions to be easily specified and greatly simplifies the precedence of operators. For example, the additive operators in JavaScript are shown in Listing 1.

```
1  public class JS_AddExpr extends PrecOperator
2  {
3    public JS_Expr left = new JS_Expr(Prec.ATLEAST);
4    public JS_Punct operator = new JS_Punct("+", "-");
5    public JS_Expr right = new JS_Expr(Prec.HIGHER);
6  }
```

Listing 1. Simplified JavaScript Additive Expression

The crucial piece is the `Prec.ATLEAST` parameter. This means the "left" part cannot have a higher precedence (e.g., multiplication).

Listing 2 shows an example of a trivial ANTLR grammar[3].

The typical output from existing tools is called an Abstract Syntax Tree (AST). It is a data structure that contains syntactic details about all the tokens found in the original computer program. For example, the additionExp node (in Listing 2) in an AST might contain three child nodes, two multiplyExp nodes and a '+' literal, to represent an addition operation. Given an additionExp node, the child nodes can be examined to determine the operator and operands. Observe that all nodes are essentially anonymous – they are identifiable by type and value, not name.

```
1  eval : additionExp ;
2
3  additionExp : multiplyExp ( '+' multiplyExp |
4                              '-' multiplyExp )* ;
5  multiplyExp : atomExp ( '*' atomExp |
6                          '/' atomExp )* ;
7  atomExp : Number | '(' additionExp ')' ;
8
9  Number : ('0'..'9')+ ('.' ('0'..'9')+)? ;
10
11 WS : (' ' | '\t' | '\r'| '\n') {$channel=HIDDEN;} ;
```

Listing 2. Sample ANTLR Grammar Excerpt

Rather than an AST, we produce a Programmar Semantic Tree (PST). It is an instance of a Java class, rather than a syntax tree. It also contains syntactic details about all the tokens in the original computer program. However, all those details are stored as instances of other objects, rather than just as anonymous nodes in a tree. For example, given an instance of JS_AddExpr called expr, the operands are expr.left and expr.right (in Listing 1).

Interestingly, the PST (output from the Parser) is stored in precisely the same classes as our grammars (input to the Parser).

2.2 LLVM Compiler Infrastructure

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies[1]. It contains a very wide array of tools focused on parsing, compiling and optimizing many different programming languages.

Our research efforts are ultimately intended to assist with programming language modernization, converting old software to new languages. We intend to do software analysis even in the absence of appropriate compilers and interpreters, and we also process non-traditional languages such as bash, sql and html. There are many areas of shared interest, so we anticipate collaboration in the near future.

2.3 Shared Language Grammars

The lack of standard grammars has made it difficult to tackle modernization problems. Analysis tools depend directly on the parse tree, which depends on the grammar used. If two researchers use different grammars, their analysis tools will probably be incompatible.

Furthermore, the grammars are typically written as text files while the tools are written in a programming language such as C++ or Java. Given the semantic gap between text and code, changes made to the textual grammar are difficult

[3] stackoverflow.com/questions/1931307/antlr-is-there-a-simple-example (condensed)

to detect in the corresponding tools. Moreover, grammars are usually under continuous change to accommodate new source code, as well as possible new versions of the language itself, so the two can become out-of-sync with no easy way to enforce consistency.

To address these issues, our system uses Java classes to represent language grammars. We use the term Program Grammar, or *Programmar*, to indicate that the grammars are themselves represented with Java classes and fields, rather than text.

Our intention is to open source all Programmars and to provide mechanisms for the community to maintain and enhance the library of Programmars in an open and organized manner.

3 Approach

Dealing with traditional grammars suffers from scalability issues. Changes to the grammar, typically written in a variation of Backus-Naur Form (BNF), can be made without the corresponding changes to the analysis tools. This often invalidates the analysis tools with no warning. Notable exceptions are systems like Rascal[9] and Stratego[2], which use pattern matching on the language directly, without relying on the grammar. Likewise, dealing with abstract syntax trees is often tedious because there is no structure other than a searchable tree.

Our approach offers several advantages. First, all changes to the grammars that impact analysis tools will be detected by the Java compiler because the grammar and the tools are both written in Java[4]. Second, the output is a collection of Java objects with named fields, called a Programmar Semantic Tree, rather than an AST. Third, the grammars are able to exploit Java language features such as inheritance and encapsulation. We implement different versions of Python or COBOL grammars by inheritance and implementing abstract classes. Allowing JavaScript or CSS inside an HTML program is easily managed with encapsulation.

We assume that our primary user base is capable of writing simple Java classes that contain little or no logic, just data fields. Based on the previous work in [13], we are using Java Reflection[6] to represent the grammars for each programming language.

3.1 Equivalence to Traditional BNF Grammars

In this section we show how the main elements in a traditional grammar are represented in a Programmar.

Note that this applies to non-terminal nodes, because terminal nodes are handled directly in Java code. See section 3.1.2 for motivation.

[4]The approach is not limited to Java and could be applied to other object-oriented languages (e.g., C# or Python)

3.1.1 Non-terminal Nodes

A fundamental decision made was to store grammars as ordinary Java code, not as text files. Here we show a mapping between the elements found in a traditional grammar and some abstract Java classes. For example, compare Listing 3 with Listing 4. Although this approach is a little more verbose, it has a collection of advantages, as described in Section 3.3. Dealing with the old AST's was like reading XML documents, i.e., using an XPath-like expression to navigate and find the element needed. With the new PST, elements are referenced by name, and because they are in Java classes, auto-completion is available to see the available names.

```
1  cPerform ::= "PERFORM" cParagraph
2      [("THROUGH" | "THRU") cParagraph] [cPerfTimes];
3  cPerfTimes ::= cExpression "TIMES";
4  cParagraph ::= cIdentifier;
5  cExpression ::= cIdentifier | cNumber;
6  cIdentifier ::= cLetter (cLetter | cDigit | "-")*;
7  cNumber ::= cDigit cDigit*;
8  cLetter ::= "A" .. "Z";
9  cDigit ::= "0" .. "9";
```

Listing 3. Simplified COBOL `PERFORM` Verb (old way)

```
1  class COBOL_Perform extends COBOL_AbstractStatement {
2    COBOL_Keyword PERF = new COBOL_Keyword("PERFORM");
3    COBOL_Paragraph startPara;
4    @OPT COBOL_PerformThrough through;
5    @OPT COBOL_PerformTimes times;
6    class COBOL_PerformThrough extends TokenSequence {
7      COBOL_Keyword THRU = new COBOL_Keyword("THRU",
8                                             "THROUGH");
9      COBOL_Paragraph endPara;
10   }
11   class COBOL_PerformTimes extends TokenSequence {
12     COBOL_Expression number;
13     COBOL_Keyword TIMES = new COBOL_Keyword("TIMES");
14   }
15 }
```

Listing 4. Simplified COBOL `PERFORM` Verb (new way)

In Listing 4, lines 2-5 are equivalent to lines 1-2 of Listing 3. Both give a high-level view of what a (simplified) `PERFORM` verb looks like. The class `COBOL_PerformThrough` is defined on lines 6-9 and used on line 4. `COBOL_Paragraph` and `COBOL_Expression` are classes defined and used elsewhere. `COBOL_Keyword` is a terminal node, based on standard library classes. The `@OPT` annotation means that the current element is optional. Table 1 provides a direct comparison.

3.1.2 Terminal Nodes

In our system, terminal nodes are implemented directly in Java code. Generally, this involves calling a shared method. It is perhaps surprising, to somebody who hasn't worked on many traditional grammars, that the terminal nodes are often quite difficult to define. Listing 5 shows a sample grammar for a Java floating point number. Limiting the value of the exponent, for example, is difficult to do, especially if the limit depends on the `float_suffix` value.

BNF Term	Eagle Abstract Class	Description
none, can use `()`	`TokenSequence`	Ordered list of tokens to match. Use `@OPT` for optional tokens.
`|`	`TokenChooser`	Alternation. Can match any one of them.
`*` or `+`	`TokenList`	Sequence of one or more tokens of type `T` (`@OPT` to allow none)
`[]` or `?`	`@OPT` annotation	Optional element
not available	`PrecedenceChooser`	Handles precedence, such as multiplication before addition.
not available	`SeparatedList<T,P>`	Sequence of elements, separated by punctuation, such as a comma.

Table 1. Comparison of BNF Terms and Eagle Abstract Classes

```
1  float_literal ::=
2    ( digits "." [ digits ] [ exponent ] [ suffix ] )
3    | ( "." digits [ exponent ] [ suffix ] )
4    | ( digits [ exponent ] [ suffix ] )
5  digits ::= "0..9" { "0..9" }
6  exponent ::= "e" [ "+" | "-" ] digits
7  suffix ::= "f" | "d"
```

Listing 5. Grammar for a Java Floating Point Number

It turns out that terminal nodes are often similar between languages. String literals generally use single or double quotes, optionally allow doubled quotes, optionally have an escape character, etc. Listing 6 shows the implementation of Java numbers (other than hexadecimal constants). The three parameters are: hex prefix characters, exponent characters, and suffix characters.

```
1  public class Java_Number extends TerminalNumberToken {
2    @Override
3    public boolean parse(EagleFileReader lines) {
4      return genericNumber(lines, "x", "Ee", "LlFfDd");
5    }
6  }
```

Listing 6. Programmar for Java Numbers

3.1.3 Context-Sensitive Grammars

When parsing and analyzing COBOL or PL/I, it is easy to simply accept a level number as a number and just store it into the AST. We have chosen to structure data records with level numbers such that they are stored hierarchically, which significantly simplifies analysis tools.

```
1  01 WS-DB-LINE.
2     03  WS-TOP-LNE.
3         05  WS-TCHR PIC X(01) OCCURS 80.
4     03  WS-T-LINE REDEFINES WS-TOP-LNE.
5         05  FILLER  PIC X(01).
6         05  WS-H-LINE
7                     PIC X(78).
8         05  FILLER  PIC X(01).
```

Listing 7. Sample COBOL Data Showing Level Numbers

Listing 7 shows how the COBOL 05 levels are inside the 03 levels, which are inside the 01 level called WS-DB-LINE.

Parsing Python is difficult to do in a context-free grammar, without pre-processing to inject markers for line breaks and indentation. Our system uses context-sensitive parsing to correctly align blocks and sub-blocks of Python code, with no pre-processing.

Because all Programmars are written in Java, the parsing routines have access to the partially built parse tree in memory. This provides context-sensitive parsing.

3.1.4 Operator Precedence

Operator precedence is handled by the `PrecedenceChooser` class. The `@P(n)` notation indicates operator precedence. In this case, it is reversed, i.e. lower precedence numbers are processed before higher numbers. Listing 8 shows the additive and multiplicative operators in C.

```
1  public @P(320) class C_MultExpr extends PrecOperator
2  {
3    public C_Expr left = new C_Expr(this, Prec.ATLEAST);
4    public C_Punct operator = new C_Punct("*", "/", "%");
5    public C_Expr right = new C_Expr(this, Prec.HIGHER);
6  }
7
8  public @P(330) class C_AddExpr extends PrecOperator
9  {
10   public C_Expr left = new C_Expr(this, Prec.ATLEAST);
11   public C_Punct operator = new C_Punct("+", "-");
12   public C_Expr right = new C_Expr(this, Prec.HIGHER);
13 }
```

Listing 8. Some Simplified C Expression Operators

3.2 Handling Multiple Languages

An important goal for our platform is to be able to analyze as many different programming languages as possible with a unified framework. This also includes the ability to analyze embedded languages, like PHP or Javascript in HTML. We also support variations of each language, such as Python 2.7 versus 3.x.

3.2.1 Standardized Parsers and Analysis Tools

When dealing with multiple languages, analysis tools can be hampered if the parse results are not stored in a consistent manner. Control flow and data flow analysis are quite similar across languages and tools should be written in a language-neutral way, whenever possible. For this reason, we have introduced a collection of Java interfaces, such as Abstract Expression, Abstract Statement, etc. that each language can implement.

For example, our transformation tool suite is based on the idea that source languages are much more varied than target

languages. We shield the transformation process from the implementation details of the target languages. The same core transformation logic is used whether the target is Java or Python or C#, as shown in [14].

3.2.2 Embedded Languages

A typical HTML file can contain Javascript, CSS, PHP or a number of other languages. PHP is actually layered on top of Perl. These compound languages are often difficult to deal with in a traditional BNF-like grammar, but are relatively simple in a Programmar.

```
1  public class HTML_ScriptBody extends TokenChooser {
2    public @CHOICE @SYNTAX(Django_Syntax.class)
3                   Django_Control django;
4    public @CHOICE @SYNTAX(Javascript_Syntax.class)
5                   Javascript_Program javascript;
6  }
```

Listing 9. HTML Script Definition (condensed)

Listing 9 shows how Django and Javascript can be embedded within an HTML file. The `@SYNTAX` notation is crucial because the syntax for each language is potentially different; languages differ on case-sensitivity, end-of-line processing, comments, etc. These syntactic differences are managed via the abstract `EagleSyntax` class. It can be compared to a "lexer", such as FLEX [11], which is often used with Bison, except it does not actually read any input characters.

3.2.3 Language Variations

Languages evolve over time. Java, for example, added generics in 2004 with JDK 1.5. One approach to handling these enhancements is to add them to the core version of the language, even if earlier versions did not support them. With software analysis, rejecting invalid programs is typically not important. If a JDK 1.4 Java program used generics, we might successfully parse it, but it would not compile. This approach is generally used in our system because it reduces the need to consider every release separately.

Sometimes, however, the changes are too significant and require a new language variant. For example, there are both free-format and fixed-format versions of RPG[7]. Although they are parsed differently, the core elements are the same and should be shared. Software analysis tools should still be able to process both formats without consideration of syntactic details.

Given two variations of a language, such as Python 2.7 and Python 3.x, it is generally necessary to create two top-level languages and have a shared abstract class that they both derive from. Since our system is written in Java, and runtime information on Generics is erased in Java[5], we have had to resort to solving the "Robot's Leg" problem using class replacement. For example, Python 2 has a 'print' statement, but Python 3 does not. It should be parsed as a statement in Python 2 and as a function in Python 3.

3.3 Scalability

Our system is specifically designed to handle millions of lines of code spanning thousands of files in dozens of languages. Accommodating this means having powerful debugging tools, reporting tools, dashboards, logging, etc. Developing such tools is greatly simplified in our system by using the same parser for all languages, implementing standard interfaces (see Section 3.2.1), and by sharing terminal nodes.

3.3.1 Debugging Tools

We have an interactive debugger that supports step-over, step-into and step-out of a parse-in-progress. This is extremely useful to monitor the progress of a file that does not parse correctly.

Figure 1 shows a screen snapshot of our interactive debugger. The core parser is independent of any debugging tools, so alternative debuggers can be created.

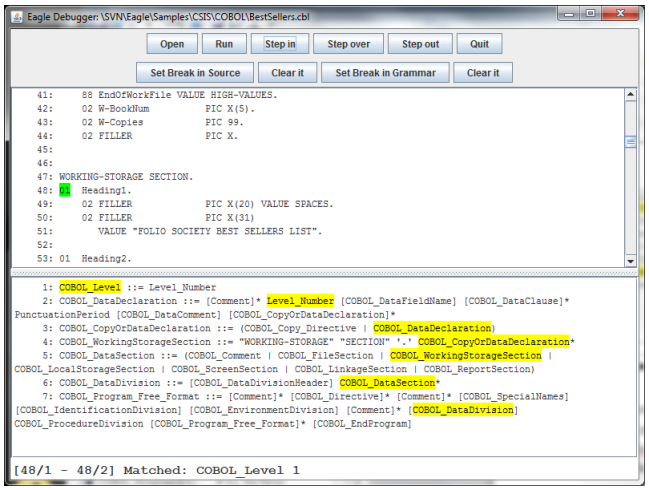

Figure 1. Interactive Debugger

In addition, we have tracing tools that monitor parsing progress. Since the parser can backtrack and re-parse on failure, these traces can often be quite lengthy. So we provide two different forms of tracing, one as a text file, and the other as an interactive HTML report with controls to open and close logical blocks of parse steps.

3.3.2 Reporting Tools

We provide a web-based tool for examining overall parse progress. It includes information about how many files didn't parse successfully, as well as information about where and why they failed. Statistics include parse speed, parse tokens per line, parse steps per line, etc. Collectively, they provide a reasonable measure of program complexity (akin to function points[4]). Figure 2 shows a sample report.

[5]https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

Additionally, reports are available for each language, including a hyper-linked BNF-style grammar, with the count of each token actually used by the test cases. This can be especially useful for deciding which language patterns to transform with automation versus manually.

Language	Files	Parsed	Bytes	Lines	Tokens	Tokens / Line	Elapsed (ms)	Lines / Sec	Parsing Steps	Steps / Line
AWK	22	22	10,934	475	8,194	17.25	629	755	45,701	96.21
CMD	73	73	61,391	1,595	22,959	14.39	733	2,175	88,606	55.55
CSS	1	1	243	19	113	5.95	0	-	1,017	53.53
HTML	3	3	14,823	217	8,550	39.40	125	1,736	68,684	316.52
Java	20	20	215,057	6,696	109,000	16.28	3,279	2,042	1,389,210	207.47
PHP	42	42	160,409	5,360	73,070	13.63	7,446	719	3,158,992	589.36
SQL	1	1	5,667	142	2,170	15.28	1,155	122	7,280	51.27
Total (7)	*162*	*162*	*468,524*	*14,504*	*224,056*	*15.45*	*13,367*	*1,085*	*4,759,490*	*328.15*

Figure 2. Example Project Summary View

3.3.3 Browsing Tools

For statically scoped languages, it is usually possible to connect symbol references with their definitions. Clicking on a definition shows all the known references, and vice versa. Syntax highlighting is automatic and controllable by a supplied Cascading Style Sheet (CSS). By default all numeric terminal nodes show up in red, key words in green, etc, regardless of programming language.

4 Discussion

A surprising number of benefits showed up with this approach to parsing and analyzing large scale software repositories. Encapsulation (Section 3.2.2), inheritance (Section 3.2.3) and context-sensitive grammars (Section 3.1.3) have already been described.

Modularity Typical BNF-like grammars are stored in a single text file. With Programmars, each language definition is split apart into logical packages and classes. For example, the Java Programmar is implemented in 46 source files, with an average of about 45 total lines per file. `Java_Subscript`, `Java_Class`, `Java_SwitchStatement`, `Java_HexNumber` are 4 examples of the 46 files.

Maintenance One of the major issues with analysis tools is when the tools get out of sync with the grammars. When one change is to a text file (e.g., BNF) and another change is in software, there may be no obvious indicators. The analysis tools may just stop matching patterns in the AST. With our approach, the tools are compiled in the same environment as the Programmars, which generally prevents them from getting out of sync.

Testing The tool suite is built in Java, and is well-suited for unit testing. We use JUnit extensively for testing low-level functions as well as large-scale functional testing.

Source Code Generation Since the Programmars are written in Java, we have added formatting annotations, such as `@NEWLINE` and `@NOSPACE`, to the Programmar for code generation. This is used for both re-formatting code and as the output from transformation tools.

5 Conclusion and Future Work

By providing a large collection of program grammars and related tools as open source, it is our hope that the community can be more successful in analyzing and modernizing legacy applications, as well as migrating modern languages to newer revisions. The tools described in this research are designed for large-scale projects involving as many programming languages as possible.

Active areas of research include cross-language data flow analysis, abstract data types, and injecting code for dynamic analysis to observe program behaviors and data value ranges. We are also researching programming language transformation on a longer term schedule.

References

[1] [n.d.]. The LLVM Compiler Infrastructure. http://llvm.org. Accessed: 2019-05-02.

[2] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of computer programming* 72, 1-2 (2008), 52–70.

[3] William H Burge. 1975. Recursive programming techniques. (1975).

[4] Tom DeMarco. 1984. An algorithm for sizing software products. *ACM SIGMETRICS Performance Evaluation Review* 12, 2 (1984), 13–22.

[5] Frank DeRemer and Thomas Pennello. 1982. Efficient computation of LALR (1) look-ahead sets. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 4 (1982), 615–649.

[6] Ira R Forman, Nate Forman, and John Vlissides Ibm. 2004. Java reflection in action. (2004).

[7] Scott Hanson and Jing Li. [n.d.]. Free-form RPG support on IBM i. https://developer.ibm.com/articles/i-ibmi-rpg-support/. Accessed: 2019-03-09.

[8] Stephen C Johnson et al. 1975. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ.

[9] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. 2009. EASY Meta-programming with Rascal. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 222–289.

[10] Ralf Lämmel and Chris Verhoef. 2001. Cracking the 500-language problem. *IEEE software* 18, 6 (2001), 78–88.

[11] John Levine. 2009. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.".

[12] Andrew J. McAllister and Steven O'Hara. 2016. Toward Effective Management of Large-Scale Software. In *SER&IP*.

[13] Steven O'Hara. 2015. Programmars: A Revolution in Computer Language Parsing. In *SERP*, Vol. 15. 125–131.

[14] Steven O'Hara. 2018. Improving Programming Language Transformation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. The Steering Committee of The World Congress in Computer Science, Computer ..., 129–135.

[15] Steven O'Hara and Jeffrey Allen Wilkinson. 2017. Parser that uses a reflection technique to build a program semantic tree. US Patent 9,710,243.

[16] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL (*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices* 49, 10 (2014), 579–598.

[17] Brian Robinson. [n.d.]. Cobol remains old standby at agencies despite showing its age. https://fcw.com/articles/2009/07/13/tech-cobol-turns-50.aspx. Accessed: 2019-03-09.