Parser Generation by Example for Legacy Pattern Languages

Vadim Zaytsev
Raincode Labs
Brussels, Belgium
vadim@grammarware.net

Abstract

Most modern software languages enjoy relatively free and relaxed concrete syntax, with significant flexibility of formatting of the program/model/sheet text. Yet, in the dark legacy corners of software engineering there are still languages with a strict fixed column-based structure—the compromises of times long gone, attempting to combine some human readability with some ease of machine processing. In this paper, we consider an industrial case study for retirement of a legacy domain-specific language, completed under extreme circumstances: absolute lack of documentation, varying line structure, hierarchical blocks within one file, scalability demands for millions of lines of code, performance demands for manipulating tens of thousands multi-megabyte files, etc. However, the regularity of the language allowed to infer its structure from the available examples, automatically, and produce highly efficient parsers for it.

CCS Concepts • **Software and its engineering → Programming by example**; **Translator writing systems and compiler generators**; **Parsers**; • **Theory of computation** → *Grammars and context-free languages*; *Pattern matching*;

Keywords parser generation, engineering by example, pattern languages, legacy software, grammar inference, language acquisition

ACM Reference Format:
Vadim Zaytsev. 2017. Parser Generation by Example for Legacy Pattern Languages. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3136040.3136058

GPCE'17, October 23–24, 2017, Vancouver, Canada

1 Problem

When working in legacy analysis and renovation industry, we come across bizarre file formats with alarming regularity. It is a world where language identification cannot rely on file extensions and may require anything up to and including machine learning [20], and where dealing with a priori unknown formats has been elevated from an idle thought experiment to a routinely used job interview question [36]. In this paper, we will share a success story of handling one of such file formats, with the *pattern language* technology (terminology by Angluin [1]).

Raincode Labs is an independent company providing bespoke compiler services. One of our clients in the banking sector, which, being NDA-bound, we will have to call 𝔄, owns a multi-million line codebase, developed over decades of company growth and containing most of its business rules and IT assets. Besides COBOL and PL/I which we have learnt to handle with ease, grace and experience, the codebase contains almost 70k modules in a fourth-generation language we will call 𝔅. Even though 𝔄 has over 100 developers actively creating new software in that language on a daily basis, it has been classified as a liability for the future and scheduled for retirement in its current incarnation. We are now in the process of writing a full-fledged compiler for 𝔅 targeting the .NET Framework. When the project is completed, it will allow 𝔄 to deploy their products on commonplace hardware or modern platforms such as Azure, to write hand-tweaked components in modern programming languages such as C♯ and, most importantly, to hire young professionals otherwise frightened off by the prospect of learning an obscure dying language as the first job requirement.

The documentation of 𝔅 is partly non-existent, partly outdated and ultimately protected legally by an explicit disclaimer that only paying customers of 𝔅's current rights owner are allowed to read it. The source artefacts come in the form of five different serialisation languages that 𝔅's infrastructure exports them in. These five notations are not synchronised: only one looks like a programming language, one more is more of a markup language, another one is syntactically and conceptually close to JSON, another one to LISP, and finally there is one notation with position-based strings (think Excel in ASCII, example on Figure 1). We will call the latter notation ℭ. All five are important for the healthy functioning of the system, since they define data and

```
$$FILE 06/07/2017 23:59:59
$$FOO    ABCD     Y 06/07/2017 23:59:59 XYZ
 A 1 00010 00 0000 Y Y N Y NAMEA    NAMEB    S
 C 2 00015 02 0000 Y Y Y Y NAMEDDDD NAME EEE S
 F 5 00030 00 0020 Y N N Y NAMEG    NAMEH    S
$$BAR    EFGHKLMN Y 06/07/2017 23:59:59 N/A
 A LONGER_NAME_FOR_ENTITY               999 10.0
 A ANSWER_TO_THE_ULTIMATE_QUESTION       42  7.5
```

Figure 1. An obfuscated snippet of what a file in ℭ may look like. The codebase contains over 20 MLOC or 3 GB of "code" like this, scattered over more than 20k files.

metadata fragments that complement one another. However, the first four notations turned out to be feasible to handle with a home-grown parser generator, technically a straightforward PEG [12] implementation with backtracking and memoisation [11], which yielded sufficiently short grammar engineering times (all grammars total under 500 LOC) and adequate performance in production (I/O bound). Yet, for ℭ it was noticeably worse for the following main reasons:

- No position orientation in PEG or any other conventional parsing techniques [15]. The focus of almost all software languages since FORTRAN and COBOL on volatile positioning has resulted in a bias against hardwired positions in parsing: column information may be a bit hard to obtain and the generative technology around parsing at best does not provide help, and at worst prevents any good designs.
- Bidirectionality support is weak. We needed to incrementally [49] co-develop the data structure and its bidirectional mapping to a textual representation for reading and writing, to be able to change it frequently and to see those changes propagated everywhere painlessly.
- Error reporting, handling, as well as recovery and correction should be stronger than in mainstream cases but still not relying on ad hoc heuristics typical for handling unstructured data.
- Since the process of reverse engineering both 𝔅 and ℭ from the codebase is challenging enough, we could not afford to add any further complications to the *process*. In particular, the process of writing a grammar or a parser for ℭ is naturally incremental and includes performing little experiments to support or refute syntactical hypotheses.
- Finally, as a matter of principle we could not rely on third party products. The solution must be possible to develop, understand and debug in-house, and maintain for decades to come.

All these objectives were successfully reached with PAX (short for PAttern eXtractor, and also a Latin wordplay because this tool brought some *peace* into the project), the solution we developed. In § 2 we sketch the landscape of existing technologies that were at our disposal and provide related work points for those who would like to consider similar techniques deeper. In § 3 we describe the first part of the solution, namely the metamodel and the syntax for writing specifications for pattern languages. Three core components are defined and considered there: patterns (pieces of a string with varying structure), commitments (structural definitions for each pattern) and bindings (mappings between a committed pattern and a node in a resulting hierarchical data structure). In § 4 the next part of the solution is sketched and assessed: how to infer such a specification from a fairly large number of positive code examples? Three main problems with the purely inferred deliverable are identified and described. The next section, § 5, lists several kinds of artefacts that are generated by PAX from the specification: a parser, an unparser, data types, tests, etc. A summary of the project and the paper is given in § 6 which concludes the manuscript.

2 Theory

Program synthesis by example is an old and mature domain. It gained popularity in the 1970s, first conquering the database/query world [29, 50]. It was quickly discovered that the same approach is applicable to small program inference [8, 10, 23, 42], search queries [27], interaction/editing patterns [6, 24, 33, 47], navigation/metaprogramming [46], requirements engineering [26], synthesised proofs [17, 31], visual/concrete mappings [34], model transformation [13, 41, 44, 45], spreadsheet transformation [16, 18], program transformation [37, 38], program correction [21, 22], etc. The ones most close to our work are directions of grammar(ware) inference/acquisition [40], which can be roughly classified into six families of methods:

- *Identification in the limit* [14] builds a representation from a sequence of samples marked with their desired correctness and converges on the target language in a finite number of steps or in finite time, both unknown beforehand. The point when the inferred grammar has converged, is also impossible to determine without applying additional mechanisms such as Angluin's *telltales* [2]. These methods are the best known but the most prone to overgeneralisation. They also diverge for any grammar class from the Chomsky hierarchy, unless negative examples are provided.
- The *probably approximately correct* [43] learning model is a weaker form of identification in the limit, and carries a compromise between accuracy and certainty, allowing to converge in polynomial time for some subclasses of languages and automata. There is a plethora of automata-based methods of inference that could technically be classified into this family, starting all the way from Chomsky [9, 39], they work by imposing a number of (sometimes mystically looking for

outsiders) limitations on languages or automata, and using those in formally proving the convergence. Some of these algorithms have near-linear complexity [3] or are sufficiently effective without negative examples [5], both qualities attractive for practical applications.

- *Concept learning* [4] uses a teacher/oracle which knows about the language and can answer several kinds of questions, of which membership ("is this program grammatically correct in this language?") and equivalence ("is there a syntactic difference between these two programs in this language?") are most important for inference of software languages, since they help eventually building a language recogniser and a parser. (In this terminology, the methods from the previous category are limited to membership queries).
- *Incremental reconstruction* [32] integrates the user of the system tightly into the workflow: by providing a few examples of how code snippets map into code model elements, a semiparser [48] is generated that handles as much input as it can; the exceptional cases are then used to develop new mappings for new iterations, which continue until the code model obtained in such a way is sufficient for the task at hand (usually some form of reverse engineering and program analysis, for which precise parsing is often unnecessary luxury).
- *Parser synthesis* [25] is an idea of inferring not just automata or grammars, but grammar*ware*—language processors such as recognisers, parsers, validators, linters, formatters, etc. There have been substantial breakthroughs recently concerning inference for inputs [7, 19] or outputs [28, 35] of grammarware.
- *Compiler inference* [30] is the dream of having semantics and runtime inferred by example as well, and arriving at a fully functional compiler (or a similarly advanced language processing tool: an integrated debugger, a build system, data binding synchroniser, etc) automatically by letting a smart algorithm consume examples, comes up with hypotheses and tests them against the oracle. This is a very tempting setup which for some cases will undoubtedly work, but for now it remains a well-planned fantasy.

The algorithm we end up using in § 4, falls into the second (PAC) category but also fulfils promises of the fifth (parser synthesis) since we generate parsers, unparsers and data structures from the inferred spec, as shown in § 5. Our algorithm is vaguely based on Angluin's approach to finding patterns common to a list of strings [1, 5], without the limitation on the number of patterns per string, but with an additional limitation on column-based pattern borders. Obviously, adjustments had to be made for inferring several unrelated families of pattern languages simultaneously based on section headers (e.g., on Figure 1 lines 3–5 are written in one format and lines 7–8 yield a different, unrelated one), as well as for multiple inclusions of the same pattern within one string (which were avoided in prior work since they make the otherwise regular languages context-free or even context-sensitive).

3 Specification

Let us first assume that we need to write patterns manually (because the pessimistic design assumes that manual adjustments will be needed in any scenario). A glance at Figure 1 can help form the following hypotheses and assumptions:

- The format of $\mathfrak{C}$ is strictly line-based.
- One file contains several blocks.
- Each block has a specially formatted header.
- Lines' structure varies per block they are in.
- Some columns are filled, some right or left padded.
- Spaces separate columns as well as occur inside them.
- Columns are of different types.

We will need to be able to specify three kinds of rules: *patterns* (where each begins and ends), *commitments* (what is the structure of each pattern) and *bindings* (how to extract information from a structured pattern and how to put it back). The combination of these three provides enough information to generate the parser for $\mathfrak{C}$, the target data structures that the parser binds the uncovered structure to, and the unparser to serialise the objects back to $\mathfrak{C}$ so that they can be saved as valid artefacts compatible with the current infrastructure at $\mathfrak{A}$.

Patterns in PAX are best explained by example. For lines 3–5 from Figure 1 the pattern would look as follows:

```
_ _ _____ __ ____ _ _ _ _ ________ ________ S
A B C     D  E    F G H I J        K
```

The concrete syntax is that the first line of the pattern definition contains hard-coded literals (in this case spaces and a capital S) and contiguous underscores which signify a placeholder for the pattern. Many alternative notations were considered but rejected: the solution had to work with extremely long lines (hundreds of characters), and this setup allows for visual debugging by simply copy-pasting a failing test case next to the pattern for visual examination, such as:

```
A 1 00010 00 0000 Y Y N Y NAMEA    NAMEB    S
_ _ _____ __ ____ _ _ _ _ ________ ____ ___ S
A B C     D  E    F G H I J        K
```

Commitment without a binding specifies the internal structure of the pattern. During parsing it be recognised but not stored in any field of a data type. For instance, it could be:

```
:- D (00|99|42|  )
```

Which means that the pattern D can contain two spaces or any of the three allowed numeric values, but never anything else. For pretty-printing, the last of the given options is usually taken (since there is no binding, the actual value of D is not stored in the tree).

A binding establishes a link between the information uncovered in the source file, and the information in memory about a newly created structural object. A binding can (and usually does) contain a commitment and may contain transformation modifiers. The full list of bindings for the lines of the first block of Figure 1 is:

```
enum Type := B:(Integer/String/Boolean) [125]
int Size       := C
int Fraction   := D
int Occurrence := E
bool Code1     := F?TF [YN]
bool Code2     := G?TF [YN]
bool Code3     := H?TF [YN]
bool Code4     := I?TF [YN]
str  Name1     := J~ [A-Z]+
str  Name2     := K~
```

The first of these is an *enumeration binding*, it contains a limited list of options allowed for the pattern (given at the end of the line), and a slash-separated list of values that will be used as enumeration values in the resulting data type. Lines 2–4 contain *integer bindings*, and they do not include an explicit commitment simply because both parsing of an integer and unparsing it with proper formatting, is built in C♯ and .NET, and we are reusing it instead of crafting our own regular expressions. Lines 5–8 are *Boolean bindings*, which have two allowed values mapped as true and false values respectively. The last two are *string bindings*, the first one of which contains an explicit commitment and the second one is taken verbatim. The trailing tilde after the letter name of the pattern, means left alignment: the value will be trimmed on the right when parsing and padded on the right (by spaces for strings and zeroes for integers) when unparsing.

Such a specification with patterns, commitments and bindings, is perfectly writeable and maintainable by hand, without any inference in sight. It is also fairly easy to create and debug, since the process basically involves copying a failing case next to it and adding a few underscores to the pattern. However, hoping to process millions of lines without 100% automation is deadly for any project with limited resources such as ours.

The final PAX specification for ℭ used in production, contains 17 patterns, 115 commitments and 120 bindings. Four lines (bundles of patterns, commitments and bindings) are explicitly reused across blocks of different kinds, without this explicit reuse the numbers would have been slightly higher.

4 Inference

Luckily, the inference algorithm did not have to perform well, since essentially it was meant to run once on our side and never come in direct contact with 𝔄. The main conceptual differences algorithmically between PAX and the Arimura–Shinohara–Otsuki [5] variant of pattern language inference [1], were already listed in § 2, but one more is worth mentioning. We have augmented placeholder increase heuristics with the knowledge of the *alphabet*: for instance, if more than half of uppercase Latin characters were observed in the same position, then its specification would be auto-promoted to the entire alphabet; if two adjacent placeholders are merged, then the most generalised of them overrides the specifications for the other; patterns specified of only 0–9 digits were auto-casted to integers; patterns often beginning or ending with zeros or spaces were assigned appropriate padding, etc. For this project the heuristic worked extremely well, but it is obviously not universally applicable.

The first version of the PAX specification for ℭ was hereby inferred from the codebase, and further refined manually. There were three main problems with the inferred specification, leading to the need of refinement:

- Underspecification of commitments. There were significantly many places where even having millions of examples was not enough to infer a thorough generalisation. In particular, very long string patterns reserved for strings of, say, 40 characters, were rarely reaching 40 characters just because no line in the codebase contained a sample that used the entire allotted space. Another example is numeric fields: there were some that reserved five positions because they were meant to store numbers between 0 and 65535, but the actual values in the codebase barely reached above 100, obviously leading to misrepresentation of the upper digits as '0' constants.
- Underspecification of bindings. Consider a situation when a particular column is a separate pattern that contains only one of two characters: M and S. The automated inference algorithm has no way of knowing whether this should be just checked for commitment, or mapped to a string, a Boolean (which one is true?), an enumeration, etc.
- Uninformative names in bindings. This was a problem unsolvable not only for the inference algorithm, but also for compiler experts. Collaboration workshops were set up with domain experts from 𝔄 who shared their "conventional" knowledge on which string column is a "long name" and which one is a "system identifier"; on why "L" is an abbreviation for "display" and on "5" being a code for character type. It is moments like these that make one seriously doubt that compiler inference as envisioned by Mernik et al. [30], is ever going to happen.

The lack of overspecification problems is a consequence of using the approach of learning from positive data, and evidence that our alphabet heuristic did not violate that property of pattern learning [1, 5].

5 Generation

The *parser* generated from the PAX specification, did not use any mainstream parsing technology [15], but thanks to the simplicity of the 𝔈 notation, it did not have to. Essentially the entire parser is a straightforward nested finite state machine. At any given moment, if the next expected piece is a literal string, a substring of the appropriate length is taken from the input and compared to the constant; if the next piece is a pattern, its commitment is checked. If this step succeeds, the binding takes place. Because of this level of granularity, error handling is extremely precise and valuable for both grammar debugging and incorrect instance recovery.

Just as PAX specification was deliberately made possible to decouple from its inference, the parser was made decoupleable from the PAX specification: all the code generated by the PAX tool is smell-free, well-formatted, readable, reasonably documented code. There was no need (as of now) to decouple the parser, but for the rest of its lifetime it is maintainable as a standalone artefact, not as a product of an application of a pipeline of complex algorithms. The entire size of the 𝔈 parser is just below 3000 LOC in C♯. The low hanging fruits in optimising it without damaging readability, were also harvested: all regular expressions are precompiled and cached, string manipulations are minimised (strings in .NET are immutable), one-symbol comparisons are done with `char`, not with `string`, etc. Parsing the entire codebase of 𝔄 written in 𝔈, takes under three minutes on an average developer's laptop and is I/O bound.

In addition to producing the parser, PAX produces the data files as well: both for the enumerations (which are shared across algebraic data types, with values collected per enumeration name) and for the actual AST nodes (from bindings). There are no scientific surprises or breakthroughs in how this works, but practically automating this synchronisation was a major time saver allowing many minor tweaks to the PAX specification to be painlessly propagated to all components—literally with the click of a button. Due to .NET specifics, pretty-printing is implemented as an `override`-method `ToString()` for the proper classes and as an extension with `public static string ToString(this ...)` for the enumerations (which may not contain methods). This allows programmers using those data types to handle them the same way.

A PAX specification can also be used as a model for testing, where variations in patterns are exercised by generating a large number of examples that are guaranteed to be correct or incorrect. This was useful as a unit testing tool during early stages to eliminate a few off-by-one errors in the generated parser, and later as a regression testing discipline when implementing parser optimisations.

6 Conclusion

In this paper, we reported on a case study in parser generation by example, and explained the motivation, genesis and application results of a tool PAX (PAttern eXtractor) capable:

- using a pattern language specification composed of patterns, commitments and bindings, to generate a working parser, unparser and all accompanying data classes;
- using a codebase of examples in a particular language, to infer such a pattern language specification of that language by incrementally learning the pattern language from variations in positive data;
- using the pattern language specification as a testing model, to generate both positive and negative examples to validate the correctness of the parser and unparser.

The process followed in this project, went as follows. We have let the inference tool produce the first approximation of the language specification by analysing the existing codebase line by line. Then, we adapted it manually, relying on regression testing and refining the specification incrementally based on conventional grammar engineering principles as well as observed behaviour. After the entire codebase seemed to have been parsed correctly, we held joint design sessions with senior engineers from the company owning the codebase, to define semantics of the inferred pieces (which was crucial for bindings in the parser but also later for implementing the compiler itself).

Additional case studies are needed to make any claims about the universality of either our approach or the tool we developed. However, the presented project is somewhat nontrivial and may be amusing as an industrial application of (usually very formal and theoretical) pattern language approaches to successfully tackle a legacy software language.

The story of PAX demonstrates how to infer the compiler machinery (data structures and (un)parsers) by example from a legacy codebase. For other legacy languages with a similar fixed-position structure, it can be applied verbatim, cutting down implementation time to a few days. Legacy languages that are significantly different from 𝔈, will have to be searched thoroughly for other properties that could make inference practically feasible.

Acknowledgements

Credit goes to 𝔄's lead engineers for providing enough information to fill the gaps in the author's knowledge of 𝔅, as well as to Sumit Gulwani for discussions about this project during PLDI'17, which gave the author extra motivation necessary to finish the submission on time against all the odds.

References

[1] Dana Angluin. 1980. Finding Patterns Common to a Set of Strings. *Journal of Computer and System Sciences* 21, 1 (1980), 46–62. https://doi.org/10.1016/0022-0000(80)90041-0

[2] Dana Angluin. 1980. Inductive Inference of Formal Languages from Positive Data. *Information and Control* 45, 2 (1980), 117–135. https://doi.org/10.1016/S0019-9958(80)90285-5

[3] Dana Angluin. 1982. Inference of Reversible Languages. *Journal of the ACM* 29, 3 (1982), 741–765. https://doi.org/10.1145/322326.322334

[4] Dana Angluin. 1987. Queries and Concept Learning. *Machine Learning* 2, 4 (1987), 319–342. https://doi.org/10.1007/BF00116828

[5] Hiroki Arimura, Takeshi Shinohara, and Setsuko Otsuki. 1994. Finding Minimal Generalizations for Unions of Pattern Languages and Its Application to Inductive Inference from Positive Data. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science (STACS) (LNCS)*, Patrice Enjalbert, Ernst W. Mayr, and Klaus W. Wagner (Eds.), Vol. 775. Springer, 649–660. https://doi.org/10.1007/3-540-57785-8_178

[6] Vipin Balachandran. 2015. Query by Example in Large-Scale Code Repositories. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 467–476. https://doi.org/10.1109/ICSM.2015.7332498

[7] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 95–110. https://doi.org/10.1145/3062341.3062349

[8] Alan W. Biermann. 1978. The Inference of Regular LISP Programs from Examples. *IEEE Transactions on Systems, Man, and Cybernetics* 8, 8 (Aug 1978), 585–600. https://doi.org/10.1109/TSMC.1978.4310035

[9] Noam Chomsky and George Armitage Miller. 1957. *Pattern Conception.* Technical Report AD110076. ASTIA.

[10] Allen Cypher. 1991. EAGER: Programming Repetitive Tasks by Example. In *Proceedings of the Ninth ACM SIGCHI Conference on Human Factors in Computing Systems*. ACM, 33–39. https://doi.org/10.1145/108844.108850

[11] Bryan Ford. 2002. Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl. In *Proceedings of the Seventh International Conference on Functional Programming (ICFP)*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 36–47. https://doi.org/10.1145/581478.581483

[12] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Proceedings of the 31st Symposium on Principles of Programming Languages (POPL)*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 111–122. https://doi.org/10.1145/964001.964011

[13] Iván García-Magariño, Jorge J. Gómez-Sanz, and Rubén Fuentes-Fernández. 2009. Model Transformation By-Example: An Algorithm for Generating Many-to-Many Transformation Rules in Several Model Transformation Languages. In *Proceedings of the Second International Conference on Theory and Practice of Model Transformations (ICMT) (LNCS)*, Richard F. Paige (Ed.), Vol. 5563. Springer, 52–66.

[14] E. Mark Gold. 1967. Language Identification in the Limit. *Information and Control* 10, 5 (1967), 447–474.

[15] Dick Grune and Ceriel J. H. Jacobs. 2008. *Parsing Techniques — A Practical Guide* (second ed.). Addison-Wesley. https://dickgrune.com/Books/PTAPG_2nd_Edition/

[16] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *Proceedings of the 38th Symposium on Principles of Programming Languages (POPL)*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 317–330. https://doi.org/10.1145/1926385.1926423

[17] Masami Hagiya. 1990. Programming by Example and Proving by Example Using Higher-order Unification. In *Proceedings of the 10th International Conference on Automated Deduction (CADE) (LNCS)*, Vol. 449. Springer-Verlag, 588–602. https://doi.org/10.1007/3-540-52885-7_116

[18] William R. Harris and Sumit Gulwani. 2011. Spreadsheet Table Transformations from Examples. In *Proceedings of the 32nd Conference on Programming Language Design and Implementation (PLDI)*, Mary W. Hall and David A. Padua (Eds.). ACM, 317–328. https://doi.org/10.1145/1993498.1993536

[19] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, 720–725. https://doi.org/10.1145/2970276.2970321

[20] Juriaan Kennedy van Dam and Vadim Zaytsev. 2016. Software Language Identification with Natural Language Classifiers. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16 ERA)*. IEEE, 624–628. https://doi.org/10.1109/SANER.2016.92

[21] Marouane Kessentini, Wael Kessentini, Houari A. Sahraoui, Mounir Boukadoum, and Ali Ouni. 2011. Design Defects Detection and Correction by Example. In *Proceedings of the 19th International Conference on Program Comprehension*. IEEE Computer Society, 81–90. https://doi.org/10.1109/ICPC.2011.22

[22] Marouane Kessentini, Houari A. Sahraoui, Mounir Boukadoum, and Manuel Wimmer. 2011. Search-Based Design Defects Detection by Example. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering (LNCS)*, Vol. 6603. Springer, 401–415. https://doi.org/10.1007/978-3-642-19811-3_28

[23] Yves Kodratoff. 1979. A Class of Functions Synthesized from a Finite Number of Examples and a LISP Program Scheme. *International Journal of Computer & Information Sciences* 8, 6 (Dec 1979), 489–521. https://doi.org/10.1007/BF00995500

[24] David Kurlander. 1993. Graphical Editing by Example. In *Proceedings of the 11th ACM SIGCHI Conference on Human Factors in Computing Systems, jointly organised with the IFIP TC13 International Conference on Human-Computer Interaction*. ACM, 529. https://doi.org/10.1145/169059.169524

[25] Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive Parser Synthesis by Example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 565–574. https://doi.org/10.1145/2737924.2738002

[26] Neil A. M. Maiden and Alistair G. Sutcliffe. 1993. Requirements Engineering by Example: An Empirical Study. In *Proceedings of IEEE International Symposium on Requirements Engineering*. IEEE, 104–111. https://doi.org/10.1109/ISRE.1993.324828

[27] Vishv M. Malhotra, Sunanda Patro, and David Johnson. 2005. Synthesise Web Queries: Search the Web by Examples. In *Proceedings of the Seventh International Conference on Enterprise Information Systems (ICEIS), Volume 2*. SciTePress, 291–296. https://doi.org/10.5220/0002510202910296

[28] Mikaël Mayer, Jad Hamza, and Viktor Kuncak. 2017. Proactive Synthesis of Recursive Tree-to-String Functions from Examples. In *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP) (LIPIcs)*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl, 19:1–19:30. https://doi.org/10.4230/LIPIcs.ECOOP.2017.19

[29] Dennis McLeod. 1976. The Translation and Compatibility of SEQUEL and Query by Example. In *Proceedings of the Second International Conference on Software Engineering*, Raymond T. Yeh and C. V. Ramamoorthy (Eds.). IEEE Computer Society, 520–526.

[30] Marjan Mernik, Goran Gerlic, Viljem Zumer, and Barrett R. Bryant. 2003. Can a Parser be Generated from Examples?. In *Proceedings of the 18th Symposium on Applied Computing (SAC)*. ACM, 1063–1067.

[31] R. Mitchell and James C. McKim. 2001. Design by Contract, By Example. In *Proceedings of the 39th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*. IEEE Computer Society, 430–431. https://doi.org/10.1109/TOOLS.2001.10028

[32] Oscar Nierstrasz, Markus Kobel, Tudor Gîrba, Michele Lanza, and Horst Bunke. 2007. Example-Driven Reconstruction of Software Models. In

Proceedings of the 11th European Conference on Software Maintenance and Reengineering, René L. Krikhaar, Chris Verhoef, and Giuseppe Antonio Di Lucca (Eds.). IEEE Computer Society, 275–286. https://doi.org/10.1109/CSMR.2007.23

[33] Robert P. Nix. 1984. Editing by Example. In *Conference Record of the 11th Annual Symposium on Principles of Programming Languages*, Ken Kennedy, Mary S. Van Deusen, and Larry Landweber (Eds.). ACM Press, 186–195. https://doi.org/10.1145/800017.800530

[34] Dan R. Olsen, Brett Ahlstrom, and Douglas C. Kohlert. 1995. Building Geometry-Based Widgets by Example. In *Proceedings of the 13th ACM SIGCHI Conference on Human Factors in Computing Systems*. ACM/Addison-Wesley, 35–42. https://doi.org/10.1145/223904.223909

[35] Terence Parr and Jurgen J. Vinju. 2016. Towards a Universal Code Formatter through Machine Learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). ACM, 137–151.

[36] Raincode Labs. 2017. The Brain Challenge: Arithmetic Puzzle. (April 2017). http://www.raincodelabs.com/blog/brain-challenge-arithmetic-puzzle/

[37] Romain Robbes and Michele Lanza. 2008. Example-Based Program Transformation. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (LNCS)*, Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter (Eds.), Vol. 5301. Springer, 174–188. https://doi.org/10.1007/978-3-540-87875-9_13

[38] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 404–415.

[39] Ray J. Solomonoff. 1959. A New Method for Discovering the Grammars of Phrase Structure Languages. In *International Conference on Information Processing*. Zator Company, 285–289. http://raysolomonoff.com/publications/newgrammars.pdf

[40] Andrew Stevenson and James R. Cordy. 2014. A Survey of Grammatical Inference in Software Engineering. *Science of Computer Programming* 96 (2014), 444–459. https://doi.org/10.1016/j.scico.2014.05.008

[41] Michael Strommer and Manuel Wimmer. 2008. A Framework for Model Transformation By-Example: Concepts and Tool Support. In *Proceedings of the 46th International Conference on Technology of Object-Oriented Languages and Systems (Lecture Notes in Business Information Processing)*, Vol. 11. Springer, 372–391. https://doi.org/10.1007/978-3-540-69824-1_21

[42] Phillip D. Summers. 1976. A Methodology for Lisp Program Construction from Examples. In *Conference Record of the Third Symposium on Principles of Programming Languages*, Susan L. Graham, Robert M. Graham, Michael A. Harrison, William I. Grosky, and Jeffrey D. Ullman (Eds.). ACM Press, 68–76. https://doi.org/10.1145/800168.811541

[43] Leslie G. Valiant. 1984. A Theory of the Learnable. *Communications of the ACM* 27, 11 (1984), 1134–1142. https://doi.org/10.1145/1968.1972

[44] Dániel Varró. 2006. Model Transformation by Example. In *Proceedings of the Ninth International Conference on Model Driven Engineering Languages and Systems (LNCS)*, Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio (Eds.), Vol. 4199. Springer, 410–424. https://doi.org/10.1007/11880240_29

[45] Dániel Varró and Zoltan Balogh. 2007. Automating Model Transformation by Example Using Inductive Logic Programming. In *Proceedings of the 22nd Symposium on Applied Computing (SAC)*, Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo (Eds.). ACM, 978–984. https://doi.org/10.1145/1244002.1244217

[46] Márcio L. A. Vidal, Altigran Soares da Silva, Edleno Silva de Moura, and João M. B. Cavalcanti. 2006. Structure-driven Crawler Generation by Example. In *Proceedings of the 29th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 292–299. https://doi.org/10.1145/1148170.1148223

[47] Andrew J. Werth and Brad A. Myers. 1993. Tourmaline: Macrostyles by Example. In *Proceedings of the 11th ACM SIGCHI Conference on Human Factors in Computing Systems, jointly organised with the IFIP TC13 International Conference on Human-Computer Interaction*. ACM, 532. https://doi.org/10.1145/169059.169532

[48] Vadim Zaytsev. 2014. Formal Foundations for Semi-parsing. In *Proceedings of the Software Evolution Week (CSMR-WCRE'14 ERA)*. IEEE, 313–317. https://doi.org/10.1109/CSMR-WCRE.2014.6747184

[49] Vadim Zaytsev. 2017. Incremental Coverage of Legacy Software Languages. In *Proceedings of the Third Edition of the Programming Experience Workshop (PX/17.2)*. ACM. In print.

[50] Moshé M. Zloof. 1975. Query-by-Example: the Invocation and Definition of Tables and Forms. In *Proceedings of the First International Conference on Very Large Data Bases (VLDB)*. ACM, 1–24. https://doi.org/10.1145/1282480.1282482