Implementation of a Workflow Management System for Non-Expert Users

Bart Jongejan
NFI-University of Copenhagen
Njalsgade 136, building 27
DK-2300 Copenhagen S
bartj@hum.ku.dk

Abstract

In the Danish CLARIN-DK infrastructure, chaining language technology (LT) tools into a workflow is easy even for a non-expert user, because she only needs to specify the input and the desired output of the workflow. With this information and the registered input and output profiles of the available tools, the CLARIN-DK workflow management system (WMS) computes combinations of tools that will give the desired result. This advanced functionality was originally not envisaged, but came within reach by writing the WMS partly in Java and partly in a programming language for symbolic computation, Bracmat. Handling LT tool profiles, including the computation of workflows, is easier with Bracmat's language constructs for tree pattern matching and tree construction than with the language constructs offered by mainstream programming languages.

1 Introduction

The CLARIN-DK infrastructure[1] has a workflow management system (WMS) that transforms, annotates and analyses the user's input. The WMS combines tools, deployed as web services, in just such a way that output is produced of the type that the user asked for, or it tells the user that such output cannot be made given the input and the available tools. The user is not required to be acquainted with the tools. In principle, neither do providers of new tools need to know which tools are registered. The currently registered tools are language technology (LT) tools (e.g., OCR, lemmatisation, syntactic analysis), but the WMS welcomes any tool that can run without direct user interaction, and new file types can be added if needed. For a description of the architecture and user interface of the WMS, see (Jongejan, 2013).

In this paper we lift the veil for the way the WMS is implemented, because we think that the symbolic computation programming language that was used for most of the implementation, Bracmat, can be used in other corners of digital humanities concerned with querying and transforming (semi-)structured data.

Symbolic computation software is primarily for solving mathematical equations, but has nevertheless a wide field of application. The WMS is one of the examples illustrating this. Tool registration, computation of workflows, simplification of the presentation of workflows before being presented to the user: each of these tasks calls for the operations that are the rationale for symbolic computation software.

Before designing the WMS, we discussed scenarios for how a user could interact with our WMS. In one scenario a user would have to assemble a workflow manually. A 'wizard' would made that task easier by for example not allowing a user to insert a tool that was not compatible with the output from already selected tools earlier in the workflow. In a more advanced scenario, the 'hairdresser' scenario, the WMS would combine the tools autonomously, while a user only had to specify the required result.

The hairdresser scenario was a logical next step, since there were already workflow systems that assisted users with assembling workflows of LT tools, for example our institute's simple but successful web-based toolbox. From the implementation point of view the hairdresser scenario had the advantage that the user interface would be simpler than in the wizard scenario. All that was needed was an input form to get the specification of the user's goal and a form where the user could choose one of the proposed

[1] https://clarin.dk/

Proceedings of the Workshop on Language Technology Resources and Tools for Digital Humanities (LT4DH), pages 101–108, Osaka, Japan, December 11-17 2016.

workflows, if the WMS found more than one solution. There would be relatively little need for feedback to the user, since the user would not be able to make errors.

Both scenarios would require input and output profile information for each integrated tool, but in the hairdresser scenario this information was required to be complete, so that no workflows could be computed that, when executed, would fail to work because of an undocumented mismatch between tools. In the wizard scenario the user's expertise could supplement incomplete registered tool information.

We chose to implement the hairdresser scenario. Java was used for all interactions with other CLARIN-DK modules and Bracmat for the computation and management of the workflows. Eight person months were allotted to the implementation of the first version, which was ready in 2011.

This paper proceeds as follows. In section 2 we present Bracmat. In section 3 we discuss why symbolic computation software and especially Bracmat makes it easy to implement an automatic WMS. In section 4 we explain why an essential feature of symbolic computation, pattern matching (PM), is best done in a programming language that has PM as a language construct. In section 5 we refer to related work. In section 6 we tell how to obtain the WMS and Bracmat. In section 7 we present the conclusions.

2 Bracmat

Bracmat is a programming language for symbolic computation created[2] by the author. Bracmat was used to perform long chains of algebraic manipulations without human supervision or interaction with intermediary results. Very soon, after minor additions, Bracmat was also used for analysis and manipulation of a much wider class of complex data, such as (semi-)structured text, thereby maximally utilizing the high level programming language constructs that already were in place to handle algebraic expressions.

PM in tree structures is an important operation in rule-based systems for symbolic computation, term rewriting and theorem proving (Ramesh and Ramakrishnan, 1992). Bracmat is no exception and has an expressive PM syntax that, in a few words, is characterised by these three PM capabilities:

- Tree PM for analysing tree structured data,
- Evaluation during PM of expressions that are embedded inside patterns, and
- Associative PM to capture zero or more subtrees with a single pattern variable.

We refer to section 6 for pointers to the full documentation of the Bracmat language and to code examples. Let it be said here that there are no restrictions on what types of expressions can be evaluated while a pattern is matched against data. It is for example possible to formulate deeply nested, even recursive, embedded PM operations, or to accumulate partial results of a PM operation, or to print messages during PM to find errors in a pattern. Associative PM, a feature that we take for granted when applying *regexp* patterns to strings, allows a pattern component to capture not just one, but a stretch of zero or more elements from a list, for example a sublist of seven subtrees somewhere in a much longer list.

Bracmat has been used in several LT related projects. Here are a few examples.

Format transformation of data. Bracmat has built-in functions for reading and writing XML, HTML, SGML and JSON data, making it easy to inspect and transform such data as native Bracmat expressions. For some tools that are registered in the WMS, Bracmat transforms data from the CLARIN-DK TEI P5 format to tool specific formats (e.g., CONLL, Penn Treebank bracketed format) and back again.

Anonymisation of court orders. In software developed for a commercial publisher of juridical text, Bracmat is used for named-entity normalisation and anonymisation of digital editions of court orders. In contrast to LT tools that only accept pre-processed plain text, this software reads the source document, which is in a proprietary XML format, and transfers all lay-out to the output (Povlsen et al., 2016).

Validation of the Dutch text corpora MWE, D-COI, DPC, Lassi, and SoNaR (van Noord et al., 2013). Bracmat was used for sampling, for checking XML well-formedness, PoS-tag usage and agreement between documentation and annotation practice, and for preparation of the tables in the validation reports.

Multimodal communication in a virtual world. In the virtual world implemented in the Staging project (Paggio and Jongejan, 2005) the functionality and data structures that let the farmer agent keep track of the dialogue and that caused appropriate agent actions and speech acts were written in Bracmat.

[2]Bracmat (then: Bacmat) made its first public appearance during a colloquium at Vakgroep Informatica, (then) Rijksuniversiteit Utrecht, 12 December 1989.

3 Workflow Computation is Symbolic Computation

From the outset, we had an intuition that symbolic computation software, and especially Bracmat, could help us tremendously to implement a WMS according to the hairdresser model. Many techniques that are required in the automatic WMS are ingrained in symbolic computation: building and destroying tree structures, testing equality of tree structures, finding common factors, sorting, factorization, normalization, and backtracking.

At the core of the WMS is the automatic computation of workflows. The algorithm we chose for computing workflows is called 'dynamic programming' (Bellman, 1957). We start from the goal, finding the tool or tools that can produce the output. The input specifications of those tools constitute subgoals that are solved recursively. We reduce the computation time by storing solutions to already solved (sub)goals, so they can be re-used. This is called 'memoizing'. Dynamic programming is also used in LT, for example in the Earley algorithm (Earley, 1983) for parsing strings using a context free grammar.

The dynamic programming algorithm itself can be implemented in any programming language, but there are many related processes where the WMS benefits of symbolic computation. We only mention two: normalization of tool registration data and simplification of the presentation of candidate workflows.

3.1 Normalization of tool registration data

The registered information about a tool consists of boiler plate information and input/output specification. The boiler plate information consists of the name of the tool, the service URL of the tool, and metadata that are not needed for running a tool, such as the name of its creator. The input/output specifications are in three levels of detail: incarnations, features and value subspecifications. We briefly explain these.

Registered I/O specification for the Brill-tagger

```
  ... other tools ...
+ ( Brill-tagger
  .   (facet,(segments*tokens^PennTree.PoS^PennTree))
      (format,(flat.flat)+(TEIP5annotation.TEIP5annotation))
      (language,(en.en))
  )
+ ( Brill-tagger
  .   (facet,(segments*tokens NER.PoS^(Moses+Parole)))
      (format,(flat.flat)+(TEIP5annotation.TEIP5annotation))
      (language,(da.da))
  )
+ ... other tools ...
```

(1)

Two incarnations of the Brill tagger. The first incarnation is for English and requires as input segments and tokens. The tokens must obey the rules set for the Penn Treebank: $isn't \rightarrow is\_n't$, etc. It outputs Part of Speech tags from the Penn Treebank tag set. The second incarnation is for Danish. It also requires segments and tokens, but optionally it can also read named entity annotations (NER). The produced tags belong to the Moses tag set or the Parole tag set. Both incarnations handle either flat or TEI P5 input and output files.

At the top level are incarnations, illustrated in code example (1) as the two sets of I/O specifications for a Part of Speech tagger. The incarnations cannot be merged into one, because they differ in more than one feature: facet and language. Mixing the incarnations would suggest that the tagger could output Parole tags for English text, for example, which is not the case. For best performance and a compact visual representation, the WMS organizes a tool's I/O specifications in as few incarnations as possible.

At the second level are the already mentioned features. Features are (ideally) mutually independent: the value of a feature has no influence on the value another feature can have. The most important features are facet (or 'type of content'), file format and language. Not all features need to be specified. If, for example, a tool is not language sensitive, then the language feature does not need to be specified.

At the third level are optional subclasses of values at the second level. For example, an image-to-text tool may produce output in the file format 'RTF', with the precaution 'OCR' as subclass. If another tool's input format is 'RTF', subclass 'OCR', or just plain 'RTF', then there is a format match.

An incarnation is computed by an algorithm that is similar to partial factorization of a sum. Factorization not only creates organization that pleases the eye, but also speeds up the calculation of workflows: it takes only two comparisons to realize that the Brill tagger in example (1) does not support German, whereas a completely unfolded representation would require checking ten different incarnations[3].

How symbolic computation and PM can help to keep the tool metadata normalized is illustrated in code examples (2) and (3). The `tools` variable is a sum, where each term is an incarnation. In a sum all terms are automatically sorted and equal terms are automatically reduced to a single term.

Both examples are PM operations. A pattern is the right hand side operand of the match operator '`:`'. The pattern in (2) continues over several lines and exhibits all three PM capabilities mentioned in Section 2: It operates on **tree structured data** (the tool incarnations), is **associative** (the variables `A`, `M` and `Z` are bound to varying numbers of intervening tool incarnations), and contains an **embedded expression**. The embedded expression, itself a match operation, is the right hand side of the '`&`' operator. The pattern is also non-linear, since the pattern variable `ToolName` occurs more than once.

Spot two incarnations that can be merged

```
1   !tools
2 :     ?A
3     + (?ToolName.?Features1)
4     + ?M
5     + ( !ToolName
6       .   ?AA
7           ( (?FeatureName,?Values2) ?ZZ
8           &   !Features1
9             : !AA (!FeatureName,?Values1) !ZZ
10          )
11      )
12    + ?Z
```

(2)

"Find two incarnations of a tool that are equal except for one feature. Bind all other tool incarnations to `A`, `M` and `Z`, the values of the equal features to `AA` and `ZZ`, the name of the exception to `FeatureName`, and the differing values to the variables `Values1` and `Values2`."

Code example (3) illustrates the ease with which a rewritten `tools` list is built from smaller chunks.

Merge two mergeable incarnations

```
1       !A
2     + !M
3     + ( !ToolName
4       .   !AA
5           (!FeatureName,!Values1+!Values2)
6           !ZZ
7       )
8     + !Z
9   : ?tools
```

(3)

"Rewrite the list of tools by reducing the number of incarnations."

3.2 Simplification of the presentation of candidate workflows

If there is more than one viable workflow, the user must choose. One of the applied strategies to make this easier for the user is to minimise the amount of information that is shown to her. This is done by not showing details that are not needed to see the differences between any two workflows in the list. A maximally simplified list of workflows displays the names of the involved tools in each workflow and those feature values and feature value subclassifications that are needed to see the differences. To bring the differences to light, the similarities have to be found and eliminated in a process that dynamically constructs and gradually improves a pattern that describes those similarities. The mathematical analogue is finding and then eliminating the polynomial greatest common divisor in a number of polynomials.

[3]The first incarnation explodes to two incarnations, one for flat text and one for TEI P5 annotations. The second incarnation doubles three times: (a) with or without NER, (b) Moses or Parole tags and (c) flat or TEI P5 format.

4 Limitations to symbolic computation with general purpose programming languages

The data types that are needed for symbolic computation can be implemented in a general purpose programming language, but the PM facilities for querying such data require specialized languages.

There are several formal query languages that are designed for querying tree structured data. These query languages can be used in scripts and programs written in mainstream languages like Java.

At first sight, writing an application using both a programming language and a query language seems to combine all the advantages of the query language with all the advantages of the programming language, but there are some disadvantages that manifest themselves when queries need to be created dynamically or when they become so complex that they no longer can be expressed in the query language.

In our understanding, a query expression, or more generally a pattern, is a declarative (as opposed to procedural) depiction of constraints on data content and data structure. That depiction can, but need not, have some similitude to data that the pattern matches. Patterns are expressed in some 'pattern language' $\mathcal{P}$. If a programming language $\mathcal{L}$ is not a $\mathcal{P}$ at the same time, then the programmer must delegate PM to a library that implements a $\mathcal{P}$. While the use of libraries in many situations results in solutions with a good separation of functionality over distinct modules, this is not the case with PM. Given only input data, a PM library still lacks the instructions from the $\mathcal{L}$ program that tell the library what to do with the data, and those instructions have to be expressed in the $\mathcal{P}$ language. A $\mathcal{P}$ expression can have a very complex grammatical structure. That structure must somehow be constructed by the $\mathcal{L}$ program, but since $\mathcal{L}$ and $\mathcal{P}$ are different languages, the $\mathcal{L}$ program is blind for the full structure of $\mathcal{P}$ expressions and not able to understand their parts. The $\mathcal{L}$ program can only handle $\mathcal{P}$ expressions at a (much) lower level than the $\mathcal{P}$ library, for example at the character string level as in code example (4).

The Java code in (4) creates an object that strings characters together to form the XPath expression `/tei:TEI/tei:text/tei:spanGrp[@ana='#tokens']`. The structure observed in Java, consisting of three simple Java String objects that are concatenated, has no relation to the structure of the query. The full structure and semantics of the query is not understood before the constructed string is handed down to the XPath library and interpreted as a $\mathcal{P}$ expression.

Java code that dynamically creates and executes an XPath expression

```
1 Document xml = streamToXml(xmlFile);
2 String id = "tokens";
3 Node span = execXPath("/tei:TEI/tei:text/tei:spanGrp[@ana='#"+id+"']",xml);
```

(4)

The split between $\mathcal{L}$ and $\mathcal{P}$ creates a need for interpretation and consequently overhead that materializes as extra programming code and an extra load on the CPU, but there are more problems with separate $\mathcal{L}$ and $\mathcal{P}$ languages. Often, as in example (4), $\mathcal{P}$ expressions appear as string values in a program that is written in $\mathcal{L}$, so $\mathcal{L}$ plays the role of host language for an embedded language $\mathcal{P}$. While $\mathcal{L}$ expressions and statements are analysed and evaluated for correct syntax during compilation of the program, syntactic and other formal errors in $\mathcal{P}$ expressions are discovered at run time, if ever. This is very unfavourable, because in rule based applications the patterns are usually the real workhorses, the most difficult to get right and most in need of debug facilities. Last but not least, a critical consequence of the asymmetry between host and embedded language is that it is impossible to evaluate $\mathcal{L}$ expressions inside $\mathcal{P}$ expressions.

In section 3 we have given an example of the usefulness of expressions that are embedded in a pattern. This embedding is only possible if $\mathcal{P}$ and $\mathcal{L}$ are one and the same language, which is the case in Bracmat.

5 Related work

5.1 workflow management systems

WMS'es are made for different purposes. For technical minded researchers, there are WMS'es that are compared to the 'make' tool by their authors. Such systems are characterised by Spartan simplicity and they make it easy to run tasks repeatedly. Examples are LuigiNLP[4] and zymake (Breck, 2008). These systems run from the command line and do not have web-based components. The information about the involved tools is minimal, and can be as little as a file extension.

[4] https://github.com/LanguageMachines/LuigiNLP

The systems GATE (Cunningham et al., 2002), UIMA (Ferrucci and Lally, 2004), TextGrid (Neuroth et al., 2011), WebLicht (Hinrichs et al., 2010), Galaxy (Giardine et al., 2005), Taverna (Wolstencroft et al., 2013), Kathaa (Mohanty et al., 2016) and Kepler (Goyal et al., 2016) all have advanced graphical interfaces that make it easy for users to combine tools in viable workflows that do useful things, but they do not tell the user whether it is possible to create a workflow that has an output that fulfils the user's requirements. It is therefore a clear advantage to have knowledge of the available tools and of how they interconnect when constructing a workflow for a specific purpose with the help of these systems.

The ALPE (Pistol, 2011) model is a framework under implementation that has similarities with our approach, the aim being to automatically construct tool chains that connect input and output. ALPE is intended as an extension to systems like GATE and UIMA.

5.2 PM languages

One of the earliest attempts at devising a programming language addressing PM was COMIT (Yngve, 1958). Descendants of COMIT were the members of the SNOBOL family (Farber et al., 1964). SNOBOL had powerful string PM facilities, but a flow of control based on labels and conditional jumps that did not mix with the pattern syntax (Griswold and Hanson, 1980).

While COMIT patterns operated on data with a simple structure (tokens and their annotations in a vector representation), later implementations of PM against text either operated on character strings using *regexp* patterns, or on tree structured data using query languages such as XPath, XQuery, XSLT, Cypher and syntax description languages like TGrep2 (Rohde, 2005), Tregex (Levy and Andrew, 2006), TIGER (König et al., 2003) and TPEMatcher (Choi, 2011).

Some mainstream programming languages have PM facilities 'built in'. Perl, for example, has powerful string PM functionality and newer versions of C# have LINQ for querying structured data in an SQL-like way. Although LINQ is a great step forward, it does not deal with associative PM.

Many functional languages, such as Haskell, ML, and Scala, have a PM mechanism against tree structured data that inspects the root of the tree and adjacent tree nodes. The matching algorithm iterates over a set of patterns until a matching pattern is found. Each pattern is part of a 'case' that defines the action to take place after a match has occurred. Functional languages overcome the split between $\mathcal{L}$ and $\mathcal{P}$, but they do generally not support associative PM.

Term rewriting systems such as XQuery, XSLT, Trafola (Heckmann, 1988), Elan (Borovansky et al., 1996), Stratego (Visser, 2001), and Tregex+Tsurgeon (Levy and Andrew, 2006) scan a tree structure, searching for terms that match a given rule pattern. When a matching term is found, the transformation part of the rule is applied to that term. In most of these systems the PM mechanism itself is not fundamentally different from that in functional languages and does not support associative PM.

Associative and associative commutative PM languages (Slagle, 1974) look for multiple matches of the same pattern with the same subject by iterating over different partitions of the subject structure in sublists and subsets, respectively. Examples are tools for computer program analysis such as Maude (Clavel et al., 1998), Tom (Moreau et al., 2003), and Rascal (Klint et al., 2009). However, these languages do not overcome the split between $\mathcal{L}$ and $\mathcal{P}$.

There are not many associative PM languages for tree structured data that overcome the split between $\mathcal{L}$ and $\mathcal{P}$. The only such languages that the author is aware of are Bracmat and the functional language Egison[5]. Egison appeared in 2011, when the first version of the WMS already was implemented.

6 Availability

Bracmat and the CLARIN-DK WMS are available under the GPL 2 license and can be downloaded from GitHub[6]. Bracmat can be compiled and executed on any platform for which a standard C compiler is available, and it can be linked to Java and Python programs. For inspirational input, hundreds of tasks on the programming chrestomathy site rosettacode.org are solved with Bracmat. Included in the GitHub distribution is documentation and a technical paper that introduces those interested in LT to Bracmat.

[5] https://www.egison.org

[6] https://github.com/BartJongejan/Bracmat and https://github.com/kuhumcst/DK-ClarinTools

7 Conclusion

Construction of a workflow between an input and a goal specification is in many ways similar to solving a set of equations with a computer algebra system performing symbolic computation. Both processes consist of several steps, both processes can finally arrive at producing a tree or directed acyclic graph structure, both processes can run into dead ends and be forced to backtrack, both processes have to handle ambiguous or unspecified parts, both processes deal with PM and unification, both processes can be implemented recursively, both processes can be sped up by simplification and normalization of all expressions and by memoizing solutions for already reached subgoals, and both processes can give a multitude of results, one result, or no result at all. It is therefore logical to implement a workflow management system not directly in a low level programming language, but in a high level language that is dedicated to building tree structures and performing PM operations on such structures.

We have spelled out three desirable capabilities of a programming language with PM as language construct: PM against tree structured data, full programmatic control during PM, and associative PM. Of the currently existing programming languages very few combine all three of these advanced features.

We presented Bracmat, a programming language for symbolic computation that does combine these three capabilities. To demonstrate the usefulness of the pattern matching capabilities of Bracmat, we have given examples of practical language technology solutions that can find application in digital humanities. One of these examples is the easy to use CLARIN-DK workflow management system.

References

Richard Bellman. 1957. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition.

Peter Borovansky, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. 1996. Elan: A logical framework based on computational systems. In *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, sep.

Eric Breck. 2008. zymake: A computational workflow system for machine learning and natural language processing. In *Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, pages 5–13, Columbus, Ohio, June. Association for Computational Linguistics.

Yong Suk Choi. 2011. TPEMatcher: A tool for searching in parsed text corpora. *Knowledge-Based Systems*, 24(8):1139 – 1150.

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. 1998. Maude as a metalanguage. In *In 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier.

H Cunningham, D Maynard, K Bontcheva, and V Tablan. 2002. Gate: A framework and graphical development environment for robust nlp tools and applications. In *Proc. 40th Anniversary Meeting of the Association for Computational Linguistics (ACL)*.

Jay Earley. 1983. An efficient context-free parsing algorithm. *Commun. ACM*, 26(1):57–61, jan.

D. J. Farber, R. E. Griswold, and I. P. Polonsky. 1964. Snobol , a string manipulation language. *J. ACM*, 11(1):21–30, January.

D. Ferrucci and A. Lally. 2004. Building an Example Application with the Unstructured Information Management Architecture. *IBM Syst. J.*, 43(3):455–475, July.

Belinda Giardine, Cathy Riemer, Ross C. Hardison, Richard Burhans, Prachi Shah, Yi Zhang, Daniel Blankenberg, Istvan Albert, Webb Miller, W. James Kent, and Anton Nekrutenko. 2005. Galaxy: A platform for interactive large-scale genome analysis. *Genome Res*, 15:1451–1455.

Ankit Goyal, Alok Singh, Shitij Bhargava, Daniel Crawl, Ilkay Altintas, and Chun-Nan Hsu. 2016. Natural language processing using kepler workflow system: First steps. *Procedia Computer Science*, 80:712 – 721. International Conference on Computational Science 2016, {ICCS} 2016, 6-8 June 2016, San Diego, California, {USA}.

Ralph E. Griswold and David R. Hanson. 1980. An alternative to the use of patterns in string processing. In *Coalgebraic Methods in Computer Science, Electronic Notes in Theoretical Computer Science*, pages 153–172.

Reinhold Heckmann. 1988. A functional language for the specification of complex tree transformations. In H. Ganzinger, editor, *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 175–190. Springer Berlin Heidelberg.

Erhard W. Hinrichs, Marie Hinrichs, and Thomas Zastrow. 2010. Weblicht: Web-based LRT services for German. In *ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11-16, 2010, Uppsala, Sweden, System Demonstrations*, pages 25–29. The Association for Computer Linguistics.

Bart Jongejan. 2013. Workflow management in CLARIN-DK. In *Proceedings of the workshop on Nordic language research infrastructure at NODALIDA 2013*, volume 089 of *NEALT Proceedings Series*, pages 11–20. Northern European Association for Language Technology (NEALT), May.

Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 168–177. IEEE Computer Society.

Esther König, Wolfgang Lezius, and Holger Voormann, 2003. *TIGERSearch 2.1 User's Manual*. IMS, Universität Stuttgart, September.

Roger Levy and Galen Andrew. 2006. Tregex and Tsurgeon: tools for querying and manipulating tree data structures. In *In 5th International Conference on Language Resources and Evaluation*.

Sharada Prasanna Mohanty, Nehal J Wani, Manish Srivastava, and Dipti Misra Sharma. 2016. Kathaa: A visual programming framework for nlp applications. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 92–96, San Diego, California, June. Association for Computational Linguistics.

Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. 2003. A pattern matching compiler for multiple target languages. In *12th Conference on Compiler Construction, Warsaw (Poland), volume 2622 of LNCS*, pages 61–76. Springer.

Heike Neuroth, Felix Lohmeier, and Kathleen Marie Smith. 2011. Textgrid - virtual research environment for the humanities. *IJDC*, 6(2):222–231.

Patrizia Paggio and Bart Jongejan. 2005. Multimodal communication in virtual environments. In Oliviero Stock and Massimo Zancanaro, editors, *Multimodal Intelligent Information Presentation*, volume 27 of *Text, Speech and Language Technology*, pages 27–45. Springer Netherlands.

I Pistol. 2011. *The Automated Processing of Natural Language*. Ph.D. thesis, Ph. D. thesis,"Alexandru Ioan Cuza" University, Faculty of Computer Science, Iasi.

Claus Povlsen, Bart Jongejan, Dorte Haltrup Hansen, and Bo Krantz Simonsen. 2016. Anonymization of court orders. In *11th Iberian Conference on Information Systems and Technologies (CISTI)*. AISTI, 6.

R. Ramesh and I. V. Ramakrishnan. 1992. Nonlinear pattern matching in trees. *J. ACM*, 39(2):295–316, April.

Douglas L. T. Rohde. 2005. Tgrep2 user manual.

James R. Slagle. 1974. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *J. ACM*, 21(4):622–642, October.

Gertjan van Noord, Gosse Bouma, Frank Van Eynde, Daniël de Kok, Jelmer van der Linde, Ineke Schuurman, Erik Tjong Kim Sang, and Vincent Vandeghinste. 2013. Large scale syntactic annotation of written dutch: Lassy. In Peter Spyns and Jan Odijk, editors, *Essential Speech and Language Technology for Dutch Results by the STEVIN-programme*. Springer.

Eelco Visser. 2001. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May.

Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, Jiten Bhagat, Khalid Belhajjame, Finn Bacall, Alex Hardisty, Abraham Nieva de la Hidalga, Maria P. Balcazar Vargas, Shoaib Sufi, and Carole Goble. 2013. The Taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41:W557–W561.

Victor H. Yngve. 1958. A programming language for mechanical translation. *Mechanical Translation*, 5(1):24–41, July.