Intricate Natural Language Processing made easier with Symbolic Computation software: Pattern Matching lessons from Bracmat

Bart Jongejan
CST-University of Copenhagen
Njalsgade 140-142
DK-2300 Copenhagen S
bartj@hum.ku.dk

Abstract

The recognition of patterns in complex data is a basic need in Natural Language Processing and often delegated to existing libraries and toolkits. Nevertheless some programming is normally needed as well. To demonstrate the usefulness of programming languages that make advanced pattern matching easy, we present such a language, Bracmat, which has already been inspiring and successful in various NLP tasks, from the validation of large text corpora to automatic chaining of NLP tools.

1 Introduction

We will present here a specially devised language in which the linguist can conveniently tell the computer to do things that he wants it to do (Yngve, 1958).

COMIT, Victor H. Yngve's brainchild, was an early pattern matching programming language that operated on a series of constituents, each consisting of a symbol or a symbol and one or more subscripts. The symbols could be words or annotations. A notable descendant of COMIT was the SNOBOL family (Farber et al., 1964; Griswold et al., 1971). Pattern matching in SNOBOL4 was handicapped by high time complexity and the conceptually unsatisfactory split between a 'basic language $\mathcal{L}$' and a 'pattern language $\mathcal{P}$' (Griswold and Hanson, 1980). Icon was designed to surmount those problems. Icon had backtracking and other advanced programming constructs, but it did away with SNOBOL4's 'pattern language $\mathcal{P}$'.

Even today, very few programming languages have native support for advanced pattern matching other than regular expressions for strings. NLP that falls outside the realm of available toolkits is now commonly done with general purpose programming languages. A search for 'programming languages for computational linguistics' or 'programming languages for Natural Language Processing' using the DuckDuckGo search machine produces, roughly in order of first appearance: Perl, Java, Ruby, Lisp, Prolog, Haskell, Mercury, Python, R, Scala, Clojure, Go, C, C++, and C#. A similar list for the search string 'programming languages for physicists' mentions several Ma*something* languages that have native support for the mathematical objects physicists work with and that are not for general purposes: Fortran, C++, Mathematica, Matlab, Excel, IDL, Python, Java, Scala, Haskell, LabView, Visual Basic, C, Lisp, Ada, Scheme, Perl, Octave, Mathcad, Nastran, Fortress, OCaml, F#, Scilab, Sage, Rlab, FreeMat, JMathLib, AMPERE, Maple, Pascal, and PL/I. Ironically, well-established packages like Mathematica and Maple owe part of their success to pattern matching facilities for manipulating symbolic expressions, continuing a tradition that started with the early (1963) computer algebra system Schoonschip (Veltman, 2000)[1].

The aim of this paper is to reawaken the craving for programming languages that have powerful pattern matching facilities for the tree structures that NLP tasks abound with. We will do so by presenting an example of such a language, Bracmat (3), its limitations (4), and some of its successes (5). But first (2) we have to explain what 'powerful' means.

2 Pattern matching in degrees

Many functional languages, such as Haskell, ML, and Scala, have a pattern matching mechanism for tree structured data that inspects the head of the tree (top node or root) and adjacent tree nodes. Each matched node can be bound to a pattern variable. A complete sub-tree can be bound to a single pattern variable or matched by a wild card '_'. The

[1] Online available at http://www.nobelprize.org/

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.

Term rewriting systems such as XQuery, XSLT, Trafola (Heckmann, 1988), Elan (Borovansky et al., 1997), Maude (Clavel et al., 1998), Stratego (Visser, 2001), Tom (Moreau et al., 2003), 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. The pattern matching mechanism itself is not fundamentally different from that in functional languages. The tree scanning mechanism is not part of the pattern matching algorithm, but defined by some tree traversal strategy.

Some pattern matching languages go further and look for multiple matches of the same pattern to the same subject by iterating over all possible ways the subject structure can be partitioned. These are associative and associative commutative (AC) pattern matching languages (Slagle, 1974; Hullot, 1979), with or without neutral elements. In this group we find computer algebra systems, some tools for program analysis such as Maude, Tom and Rascal (Klint et al., 2009) and a few less specialised programming languages, such as Egison (Egi, 2014) and Bracmat, but not any languages with a user basis in the NLP community.

The formal legitimation of partitioning a list in multiple ways is that the binary operators that link the list elements can be associative and have a neutral element. Associative binary operators $\circ$, for example the algebraic $+$ and $*$ operators and concatenation, have the following property:

$$x \circ y \circ z = (x) \circ (y \circ z) = (x \circ y) \circ (z)$$

Associative pattern matching takes account of the associativity of the binary operator connecting list elements by tentatively partitioning the list in as many parts as there are pattern components and by repeating this until all partitions have been tried or until each pattern component matches a corresponding part of the list.

A binary operator with a neutral element makes associative pattern matching even more versatile. A neutral or identity element e with respect to a binary operator $\circ$ has the property that

$$e \circ x = x = x \circ e$$

With respect to the binary operators for addition, multiplication and concatenation, the neutral elements are 0, 1 and the empty string '', respectively.

If a pattern with M components has N components that can match a neutral element, then the pattern matching algorithm must not only investigate all partitions with M parts, but also with $M-1, M-2, \ldots M-N$ parts. For example, if the first and the last component in a pattern with three components can accept neutral elements, then the pattern matcher, considering partitions with 3, 2 or 1 parts, has to try the following partitions of a list with only two elements x and y:

$$(e) \circ (x) \circ (y)$$
$$(e) \circ (x \circ y) \circ (e)$$
$$(x) \circ (y) \circ (e)$$

The combination of tree pattern matching and associative pattern matching with neutral elements is very powerful and useful, as we will show here.

3 Bracmat

Bracmat is a symbolic computation program, originally conceived with the aim of performing long chains of algebraic manipulations without human supervision or interaction with intermediary results. Later additions made it possible to use Bracmat for analysis and manipulation of a wider class of complex data, maximally utilising the high level programming features that already were in place to handle algebraic expressions.

Bracmat must evaluate expressions to a form that is normalised and that it cannot further simplify, in order to avoid that computations in later steps become inextricably complicated and perhaps even impossible to perform. Canonisation of expressions is to some degree achieved by the hard coded expression evaluator, which sorts and merges sums of terms and products of factors[2] and applies basic simplification rules to pairs of juxtaposed elements:

$$(b + a) + (a + -b) = (a + b) + (a + -b) =$$
$$a + (a + (b + -b)) = 2a + 0 = 2a$$

Because concatenation is not commutative, concatenated lists are not merged, but flattened:

(The sun) (is red) = The (sun (is red))

[2]This is Associative Commutative (AC) normalisation.

Bracmat's expression evaluator normalises trees constructed with addition, multiplication and concatenation operators to right descending trees without neutral elements.

The expression evaluator cannot simplify an expression if it doesn't create or find pairs of elements that can be combined, such as in this case:

$$\frac{b+x+by+xy}{1+y}$$

A general algorithm that can simplify this expression is hard to implement, test and debug if directly written in C, the language in which Bracmat is implemented. Therefore Bracmat is a programming language in its own right, with a relatively simple and thoroughly testable pattern matching facility as its most important high level language construct. Associative pattern matching with neutral elements enables Bracmat to reason about an expression in its entirety.

Bracmat lacks commutative pattern matching, but this should not be seen as a shortcoming. Because Bracmat normalises expressions before subjecting them to pattern matching, patterns should always mimic normalised expressions. For instance, a pattern component to capture a numerical factor must be the first factor in a product of pattern components, as this is the only position where numerical factors can occur after normalisation.

Bracmat is a homoiconic language like Lisp: there is no distinction between data and program instructions. Bracmat expressions are immutable[3] binary trees built from binary operators[4], leaf nodes (or 'atoms'), which are strings, and prefixes[5] attached to those leaf nodes or binary operators. The illusion of expression evolution is created by disassembling existing expressions and assembling new ones. First, by using pattern matching, all parts of the expression that should be reused are retrieved. Then, a new expression is composed from the retrieved parts, together with new pieces, as shown in this Bracmat example:

```
  my dogs run:?D dogs ?V
& do !D cats !V too "?"
```

This expression evaluates to:

```
do my cats run too ?
```

[3]Exception: the rhs operand of an = operator is replacable.

[4]`= . , | & : space + * ^ \L \D ' $ _`

[5]`[ ~ / # < > % @ ` ? ! !! -`

The binary operator `:` is the pattern matching operator. Its left operand is the subject of the operation, a list of three symbols (string tokens) linked by two space (concatenation) operators[6]. The right operand `?D dogs ?V` is a pattern with three components linked with space operators.

Pattern components consisting of a symbol without prefixes (`dogs` and the literal question mark `"?"`) match only that symbol. Pattern components with `?`-prefix can capture values. The prefixed expressions `?D` and `?V` are pattern variables. Each time there is a match the captured value is bound to the pattern variable. No assignment takes place if the prefixed expression is an empty string.

The second line of the expression starts with `&`, meaning 'and then'. This operator, and likewise the 'or else' operator `|`, have short-cicuit semantics. When the left operand of a `&` operator (the first line in this case) succeeds, the right operand (the remainder of the second line) is evaluated and returned. If the left operand fails, the whole expression fails.

The expressions with `!`-prefix, `!D` and `!V`, are the same pattern variables as `?D` and `?V`, but in a different role: they produce the values that are bound to those variables.

A pattern is said to be non-linear if a pattern variable is capturing and later during the same pattern matching operation producing. The following example shows how we find two inventions `i1` and `i2` in the same year `yr` in a list of six (*name.year*) pairs using a non-linear pattern. The dot operator is a binary operator that is the cement in many fixed data structures. Observe that a solitary `?` is a wild card.

```
(   (penicillin.1928)
    (sonar.1906)
    (teabag.1904)
    (telephone.1876)
    (triode.1906)
    (zeppelin.1900)
  : ? (?i1.?yr) ? (?i2.!yr) ?
& !yr saw !i1 and !i2
)
```

Evaluation of this expression returns:

```
1906 saw sonar and triode
```

[6]Any number of white space characters between two subexpressions corresponds to one space operator. The white spaces in `x + y` are not white space operators, because + is not an expression.

The pattern component `!yr` always evaluates to the latest value captured by the `?yr` pattern component.

Without going into the details of factorisation, an outline of a solution to the factorisation task discussed earlier looks like this expression:

```
    ( fct = local vars . body )
  & fct$((1+y)^-1*(b+x+b*y+x*y))
```

The first line of the expression declares and defines a recursive function `fct` that visits and factorises all sum nodes. The = operator assigns its right operand to its left operand without first evaluating the right operand. Local variables are declared to the left of the dot operator, while the right operand is the function body, where pattern matching and expression rewriting takes place. The actual argument expression to a function (always called `arg`) needs no declaration.

When Bracmat is asked to evaluate the expression, it first defines the function `fct` and then applies, using the function application operator `$`, this function to the expression to factorise, returning the expression

```
(1+y)^-1*(1+y)*(b+x)
```

Because Bracmat's expression evaluator inexorably combines the factors with the common base `(1+y)`, this immediately becomes `b+x`.

Bracmat can not only apply patterns to tree structures, but also to a single leaf node, which always is a string of zero or more characters. Syntactically there is almost no difference between string pattern matching and tree pattern matching in a list with space operators:

```
c l a s s:? a s ?   { tree PM }
@(class:? a s ?)    {string PM}
```

The `@` ('atom') prefix is attached to the `:` operator, which heads the parenthesised expression. It tells Bracmat to use string pattern matching instead of the default tree pattern matching.

4 Limitations and solutions

Some of those who stood at the cradle of SNOBOL4 mention eight reasons to do away with pattern matching (Griswold and Hanson, 1980). Of these, the following issues are relevant in the case of Bracmat:

The complexity of the pattern matching algorithm. Pattern optimisation makes it hard for a programmer to come to grips with the complexity of SNOBOL4 patterns. Bracmat patterns are not compiled or optimised and fairly predictable, but some experimentation is sometimes necessary.

Unnecessary backtracking and lack of control over the pattern matching algorithm. An example of this is the following:

```
    ( p = | x !p (x|y) )
  & x x x x x x x y x x : !p
```

The pattern `p` matches the subject if either the subject is an empty string or the subject starts with `x`, followed by a sequence that matches the value of `p`, followed by a single `x` or `y`. In other words, pattern `p` is recursive and matches lists of N `x`s followed by N `x` or `y`s, $N \geq 0$.

The pattern `p` is terrible inefficient, because the pattern component `(x|y)` does not find a match before the much more complex pattern component `!p` has matched the $2(N-1)$ elements between the first and the last element. Bracmat patterns are non-greedy, so `!p` is first matched against the empty string, then with one `x`, two `x`s, three `x`s, ..., forced by backtracking when either `!p` fails (every second time) or the sub-pattern `(x|y)` fails. This process repeats at each level of recursive invocation of the pattern `!p`.

To write an efficient pattern, the programmer should economise with backtracking and recursion, for example by postponing backtracking and recursion until the easiest and cheapest components are matched. Reformulated in this vein, the problem is: make a pattern that either matches the empty string or that matches a subject that begins with an `x` and that ends with an `x` or a `y`, and that also matches the list between the first and the last elements:

```
  ( q = | x ?V (x|y) & !V:!q )
& x x x x x x x y x x : !q
```

Because Bracmat does not optimise patterns, it is the programmer's responsibility to write procedurally efficient patterns.

Difficulties with program structuring, especially the necessity for using side effects. In Bracmat, side effects cannot be eliminated, but they can be restricted to locally declared variables inside functions.

5 Examples

5.1 Medical question answering

The ESICT platform (Henriksen et al., 2014) was aimed at answering questions from Danish diabetes patients. Questions were tokenised, lemmatised, syntactically parsed, analysed for relations between medical terms and then coded as queries to the SNOMED CT terminology bank. Bracmat was used for lemmatisation and for relation extraction, using hand-made syntactic patterns. It was not feasible to use statistical methods due to the paucity of example questions.

5.2 Multimodal communication in a virtual world

The Danish Staging project has brought forth a 3D virtual world inhabited by a farmer agent and his animals. The user, through an avatar, can communicate with the farmer using natural speech and hand movements. The functionality that lets the farmer keep track of the dialogue and that generates appropriate agent actions and speech acts was written in Bracmat (Paggio and Jongejan, 2005). A rapid development of this module was made possible due to the ease of saving, inspecting, editing and hot deploying Bracmat script files.

This application is characterised by a number of data structures that together embody static and dynamic world knowledge: the scoreboard that keeps track of salient objects, the evolving dialogue tree, and a table that lists which types of response to expect given some dialogue initiative. Incompatible responses push the current initiative on a stack and initiate a sub-dialogue. When the sub-dialogue has been completed, the initiative is popped from the stack.

Changes in all these data can be traced in real time.

5.3 Rewriting HTML pages

Bracmat was used to parse, repair, modernise and validate about 1000 manually written HTML files. The files had many syntax errors and deprecated elements. A Bracmat script repaired broken elements, rewrote deprecated elements to allowed elements, removed redundant elements, and restructured the HTML tree if elements occurred in places where they were not allowed. The result was validated with W3C's XHTML validator. A report was generated of the whole process so it was possible to see which files still had issues. Only a few files with exceptional errors had to be corrected by hand. As a simple example, let us assume that we want to change the HTML attribute

`clear="`*XX*`"`

to

`style="clear:`*XX*`;"`

where *XX* is an unknown value. It is possible that the `style` attribute already exists and even that it already has a `clear:`*XX;* value.

In the following Bracmat code snippet, `!attrs` evaluates to the list of `(attribute.value)` pairs of the element currently in focus. The variables `First` and `Last` match an capture any attributes preceding or trailing the `clear` attribute.

If a `style` attribute with a `clear:`*XX;* value is already present, nothing will happen except that the HTML attribute `clear:`*XX;* is removed. If a `style` attribute exists, but without a `clear:`*XX;* value, the existing value `W` of the `style` attribute is concatenated with that new value using the the `str` function. Otherwise a new `style` attribute value is created. The original `clear` attribute is not returned in the new value of `attrs`. The whole expression fails if there is no HTML `clear` attribute.

```
  !attrs:?First (clear.?XX) ?Last
& str$("clear:" !XX ";"):?newVal
&    (     !First !Last
         : ?pre (style.?W) ?post
       & !pre
         ( style
         .   @(!W:? !newVal ?)
           | str$(!W !newVal)
         )
         !post
     | !First (style.!newVal) !Last
     )
  : ?attrs
```

In the actual program the attribute rewriting code is put in a generic function.

5.4 Dynamic programming

The CLARIN-DK infrastructure has a workflow service that assists users with analysing and annotating language resources, even if they do not have much knowledge of NLP tools (Jongejan, 2013). Unlike other language resource tool frameworks (Ogrodniczuk and Przepiórkowski, 2010), which

assist the user in selecting NLP tools as workflow components, the CLARIN-DK toolbox just asks the user to specify the desired end result. It then presents a list of suitable workflows and runs the one the user selects. An empty list of workflows indicates that no solution, man made nor machine made, exists, given the registered tools in CLARIN-DK.

The workflow service, employing top down, recursive dynamic programming with memoisation, computes workflows that, starting from the provided input, satisfy the user's goal. It does so by trying to find the NLP tools in its registry that can produce output as stated in the goal. For each tool found, the service recursively tries to satisfy the input requirements of that tool, until those input requirements are met by the input or until no tools are found that satisfy the input constraints or until a recursion depth limit (currently set at 20) is reached. After finding a solution, the workflow planner backtracks until a choice point to try an alternative path, which may lead to another viable workflow. This is repeated until the workflow service has found all (perhaps zero) workflows.

When describing their goal, users are advised to specify only features that matter to them, because a too detailed specification can lead to an empty solution set. There are currently just four features (type of content, presentation, format and language) that can be specified. Features can have values independent of each other. For example, a musical or literary work or an annotation can be fixed in almost any format, e.g., LaTeX, GIF page images or a video stream.

Computer algebra functionality is critical for the success of the workflow service. Many workflows are not linear sequences of tools, but are better described as directed acyclic graphs, because many NLP tools take two or more inputs, for example a text and one or more annotation layers to that text. If a tool has T independent inputs that need to be generated in earlier steps, then there are $T!$ ways to arrange the steps generating those inputs in a job queue. All these arrangements are equivalent. The workflow service, employing the automatic transformation of polynomials to normal form, designates one of the $T!$ queue orders as the canonical order. This is extremely important, because the (in-)equality of two workflows can only be established if both are normalised to unique canonical forms.

Many tools allow some flexibility in the input given and in the output produced. For example, a tool may be able to handle a number of languages, or it can take a number of different file formats. Or it can do both. Some tools can even (perhaps as an option) take several combinations of types of input. For example a lemmatiser may be able to produce more accurate results if the word classes of the words are already known. Such alternatives are expressed as a sum of symbols, where the + symbol means 'OR'. For example, if a tool can take either Chinese, English or French input and produces output in the same language as the input, then we can express the tool's language feature as

```
( language
, (en.en)+(fr.fr)+(zh.zh)
)
```

Likewise, the content type feature of the lemmatiser that optionally can take Part-of-Speech (PoS) tags in addition to tokens, can be specified by using the $\star$ symbol to express an 'AND' relation:

```
( contentType
, (tok+pos*tok.lem)
)
```

The symbols + and $\star$ are chosen to fulfil the roles of OR and AND because they obey the same distributive law as these logical operators. An additional benefit of using these mathematical operators is that they are commutative and associative, making it straightforward to normalise expressions that use these operators.

Although the use of mathematical operators to build data structures may look strange at first, there is a close analogy with functional languages. The multiplication operator creates product types (synonyms: tuples, records), while a sum operator creates sum types (synonyms: unions, variant types). These concepts are well known in functional languages. While Bracmat does not itself have algebraic data types, it is at least very natural for a Bracmat program to reason about algebraic data types!

Another computer algebra feature is used to condense and simplify the metadata of CLARIN-DK tools. A tool that supports 12 languages and three input file formats has 12 x 3 = 36 different input combinations. It would be convenient if we could just keep talking of 12 languages and three input file formats. To be able to do this, the metadata must be kept factorised and presented to the

user as mutually independent lists of values. For a minor tool extension the registrant of the tool only needs to open the registration form that shows the current metadata and to add the new value to the appropriate feature. For bigger changes, for example the addition of a new language that requires an exceptional file format, the registrant can choose to fill out a registration form from scratch. The tool service then adds the new metadata to the existing metadata and re-factors all metadata.

5.5 Named Entity Normalisation (NEN)

A Bracmat program is part of an automated module that anonymises court orders for the Danish company Schultz Information[7]. First, a Perl program tokenises the XML input and recognises named entities (NEs), such as names of legal entities and registration numbers. Next, a Bracmat program fuzzy matches all words in the found NEs with all words in the text. This step increases the recall to almost 100 %, somewhat at the expense of precision. In following steps the NE candidates are pruned using several heuristics, until all ambiguity is removed. In the last step the Bracmat program de-identifies all NEs and replaces each NE with a referent ID, strictly adhering to Schultz's guidelines for the encoding of anonymous referent identifiers: `person1`, `person2`, `company1`, (account number) `...876`, etc. .

The ease with which test documents could be saved at each stage of the NEN process, made it easy to see the effects of each step. This made it possible to develop the complex heuristics within a relatively short time.

5.6 Corpus validation

Bracmat has been used to assist manual and automatic validation of a number of Dutch corpora (MWE, D-COI, DPC, Lassi, and SoNaR) (van Noord et al., 2013). Each validation consisted of varying tasks, such as sampling, reporting and checking XML well-formedness, PoS-tag usage, and agreement between documentation and annotation practice. The process of understanding a single validation task and simultaneously writing, testing and finally running a Bracmat script for doing the task took between an hour and a week.

Bracmat can natively read XML, SGML, HTML and JSON into Bracmat expressions. Reading mark-up is robust against non-wellformedness and schema related errors, making it possible to localise and describe each error using pattern matching.

Here is one line[8] from an informal document describing validation criteria, which turned out never to have been enforced. The validation criteria are written directly in a file that resembles the files to validate:

```
<CollectionCode>Batch/SoNaR/
n1;cc;src;n2</CollectionCode>
n1 = 1-7, cc = NL or BE, src
= A,B,C or D, n2 is 2 digits
```

Using Bracmat's option to read XML, the line was transformed to this Bracmat code:

```
(CollectionCode
.,"Batch/SoNaR/n1;cc;src;n2"
)
" n1 = 1-7, cc = NL or BE, src
= A,B,C or D, n2 is 2 digits "
```

This line was edited to become a validating pattern that issued an error message upon failure:

```
( CollectionCode
.
  ,   @( ?
       :   "Batch/SoNaR/"
           (~<1:~>7)
           ";"
           (NL|BE)
           ";"
           (A|B|C|D)
           ";"
           #%@
           #%@
       )
    | {... error message}
)
```

Inside the tree pattern for a `CollectionCode` element resides a string pattern. Where we normally expect the subject string to be, we find a wildcard `?`, because the left operand of the string match operator doubles as a pattern component in the outer tree pattern.

The pattern components `#%@` match exactly one decimal digit each. The prefix `#` ('number') matches any rational number, the `%` ('$\geq$ 1') prefix matches one or more non-neutral elements. Finally, the `@` ('atom', '0 or 1') prefix matches one

[7] http://progresso.dk/en/schultz-anonymisation/

[8] The line, which is more elaborate in the original document, is adapted for the sake of clarity of the example.

neutral or non-neutral element. As in a wild card expression `?`, the prefixed string is empty in `#%@`.

Interestingly, a small part of the corpus did in fact conform to the informal specification.

5.7 Expression embedding in patterns

A powerful feature is that inside patterns, Bracmat gives full access to the language. The `&` operator can be used to decorate a pattern component with an expression that is evaluated after each match of that component.

The following example finds and sorts the strings that occur twice in the subject:

```
( 0:?S
& (   I blink and you smile
      as I see that you see me
    : ? %@?x ? (!x &!x+!S:?S&~) ?
  | !S
  )
)
```

Let us dissect this example. The non-linear pattern

```
? %@?x ? !x ?
```

matches if a string, captured by `%@?x`, occurs twice in the subject. The `!x` component in this pattern is decorated with the expression

```
!x + !S : ?S & ~
```

This expression adds the found string to the accumulator variable `S` and then fails due to the always failing expression `~`. That forces the pattern matcher to backtrack and try to find other candidates. Finally the accumulated value bound to `S` is returned: `I+see+you`.

5.8 Projects inspired by Bracmat

Iconclass (van den Berg et al., 1994) is a classification system designed for art and iconography. The first finished version existed only in paper format (van de Waal et al., 1985). Its intricate hierarchical data structure posed insurmountable problems during the first attempts at computerisation based on database software. To test a radically different approach, a prototype of the first working IconClass browser was created in Bracmat. The data structure of the first production version in C carried the clear traces of Bracmat and survived for many years.

The handling of morphological changes to prefixes, infixes and suffixes by the lemmatiser described by Jongejan and Dalianis (2009) was mentally prototyped by experiences with Bracmat.

6 Bracmat in practice

Bracmat is freely available at GitHub and runs on every 32 or 64 bit platform for which a standard C compiler is available. To modern standards, its memory footprint is very small.

Installing and running Bracmat is uncomplicated. If started without command line arguments, Bracmat starts in interactive mode and runs a REPL (Read-Evaluate-Print Loop).

Correct indentation of Bracmat code is very important for good readability, but is of no importance for the Bracmat interpreter. It is best to not spend time on manual indentation, but let Bracmat take care of that and to visually check the indented listing. Many programming errors stand out by code being indented in unexpected ways.

Bracmat listings have a lay-out that keeps the text within about 80 columns. Most code examples in this paper are produced by using the `lst` function.

7 Conclusion

Pattern matching is a programming language construct that has been invented many times and that has died a few times as well. The reasons for its demise are not always clear, but in at least one documented case, bad or unpredictable performance was the culprit. Another reason for the lack of success may be that the advanced languages that support associative (perhaps even commutative) pattern matching with neutral elements are developed for small audiences and can have a steep learning curve for outsiders. It may be the case that tools for program optimisation can also be used in the context of NLP, but who wants to be the first one to try and perhaps fail?

What is needed is a programming language for NLP with pattern matching as a first class programming construct, and not as a 'pattern language $\mathcal{P}$' as a guest in a 'basic language $\mathcal{L}$'. This is because, as we have seen in some of the examples, access to the full language is useful even in the midst of a pattern matching operation. Associative pattern matching with neutral elements and the corresponding normalisation (flattening and neutral element removal) are already well known from string pattern matching, but should also be available when working with structured data.

Bracmat has been successfully employed in several real life applications in the field of NLP, proving the merits of such powerful pattern matching.

References

Peter Borovansky, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. 1997. Elan: A logical framework based on computational systems. Elsevier.

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.

Satoshi Egi. 2014. Non-linear pattern-matching against unfree data types with lexical scoping. *CoRR*, abs/1407.0729.

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

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.

R.E. Griswold, J.F. Poage, and I.P. Polonsky. 1971. *The SNOBOL 4 programming language*. Automatic Computation Series. Prentice-Hall.

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.

Lina Henriksen, Anders Johannsen, Bart Jongejan, Bente Maegaard, and Jürgen Wedekind. 2014. Worlds apart–ontological knowledge in question answering for patients. *Proceedings of the Fourth Workshop on Building and Evaluating Resources for Health and Biomedical Text Processing BioTxtM2014*, pages 76–83, May.

J. M. Hullot. 1979. Associative commutative pattern matching. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'79, pages 406–412, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Bart Jongejan and Hercules Dalianis. 2009. Automatic training of lemmatization rules that handle morphological changes in pre-, in- and suffixes alike. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 145–153. Association for Computational Linguistics.

Bart Jongejan. 2013. Workflow management in CLARIN-DK. In *Proceedings of the workshop on Nordic language research infrastructure at NODALIDA 2013*, number 20 in 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.

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*.

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.

Maciej Ogrodniczuk and Adam Przepiórkowski. 2010. Linguistic processing chains as web services: Initial linguistic considerations. *Proceedings of the Workshop on Web Services and Processing Pipelines in HLT: Tool Evaluation, LR Production and Validation (WSPP 2010) at the Language Resources and Evaluation Conference (LREC 2010)*, pages 1–7.

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.

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

H. van de Waal, L.D. Couprie, and E. Tholen. 1985. *Iconclass: an iconographic classification system*. North-Holland Pub. Co.

Jörgen van den Berg, Hans Brandhorst, and Peter van Huisstede. 1994. Iconographic thesaurus: A key to subject retrieval in pictorial information systems. In Alexander Tzonisian White, editor, *Automation Based Creative Design - Research and Perspectives*, pages 265 – 279. Elsevier, Oxford.

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.

Martinus J. G. Veltman. 2000. From weak interactions to gravitation. *International Journal of Modern Physics A*, 15(29):4557–4573.

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.

Victor H. Yngve. 1958. A programming language for mechanical translation. *Mechanical Translation*, 5(1):25–41, July.