UNIVERSITY OF COPENHAGEN

Implementation of a workflow management system for non-expert users

Bart Jongejan

Abstract

Language technology (LT) tools can be combined into workflows and together annotate or transform input data in ways that can be useful for many scholars. However, manually creating such workflows can be a daunting technical task. Some workflow management systems guide the user through this process, so she avoids making impossible combinations of tools. Other workflow management systems make workflow planning as easy as planning a route between two locations by means of a car navigation system. One of the elements of the initial (2008-2011) Danish contribution to CLARIN[1)] is an example of the latter. Henceforth we call this system 'the WMS'. It asks the user to specify the input and the desired output, and then it computes how LT tools can be combined into workflows that give the desired result.

Instrumental for the implementation of the WMS was a programming language specialized in symbolic computation and pattern matching, Bracmat. This poster tells why precisely this software made it possible to build the WMS in a relatively short time.

The kind of pattern matching that sets Bracmat apart from other programming languages is useful in many scholarly software applications. Hopefully, other programming languages will obtain a similar language construct for pattern matching.

1) European Research Infrastructure for Language Resources and Technology

The WMS: simple outside, complex inside

The WMS is a web service. There are several ways to submit input to this service: by upload, by URL, by selecting a resource from a repository, or as shown in Fig. 1, by typing or copying and pasting some text.

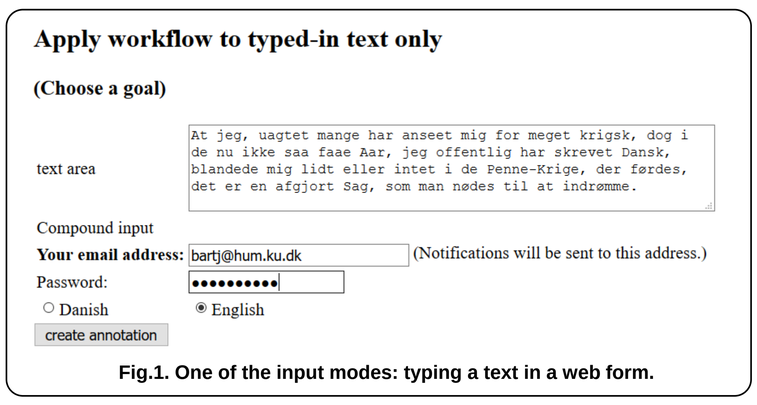

Fig.1. One of the input modes: typing a text in a web form.

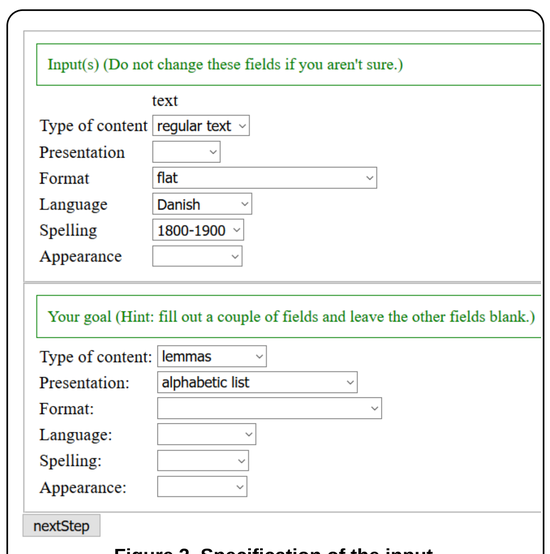

Figure 2. Specification of the input and the desired output of the WMS.

After pressing the *create annotation* button, the next screen appears (Fig. 2). Some of the form fields are already filled out, most notably the file format of the input.

The user is asked to fill out some, but not necessarily *all*, remaining fields, such as the language and the desired type of content of the output.

A few seconds after pressing the *nextStep* button the WMS comes up with the possible workflow solutions, see Fig. 3. The user then submits one of the solutions and is notified when the workflow has finished.

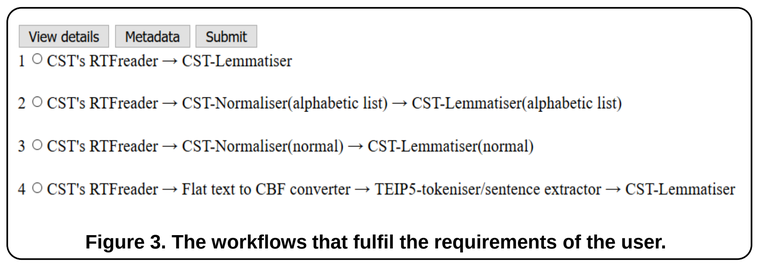

Figure 3. The workflows that fulfil the requirements of the user.

The information that is needed to compute workflows consists solely of
- the specification of the input and output of the workflow (Fig. 2) and of
- the input and output profile specifications of all tools that the WMS knows about. These metadata are obtained when tool providers register new tools. (There is a web form for that, too.)

Tool providers and the maintainers of the WMS do not need to know in which ways tools can be combined.

To minimize the cognitive load on the user, only information that highlights the differences between the workflows is shown in the list of viable workflow solutions. The reduction of detail in visualisations of workflows was one of the hardest WMS features to implement.

A workflow is internally represented by a tree structure. The tool producing the final output is in the root of the tree and the leafs are populated by the tools that take the workflow input. The other nodes represent intervening tools, while the edges (the arrows in Fig. 3) connect tools with matching output and input profiles. The examples in Fig. 3 have very simple tree structures, with only a single leaf and no branches that come together.

The choice of programming language matters

Workflows inherit much of the structural complexity of tool profiles, and add to that a structure of their own. During the creation of workflows a large amount of smaller and bigger tree structures appear and disappear because of the tentative character of some creation steps. Similar dynamic processes take place when tool metadata is registered and when workflows are simplified before being shown. This activity is typical of symbolic computation and is similar to what a computer algebra system does to rework an algebraic expression.

During all these dynamic processes the program has to search, inspect and test the currently existing data structures, so it can decide how to proceed. Querying these tree structured data is most efficiently done using a pattern matching (PM) facility of some kind.

Queries, or patterns in general, 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 $\mathcal{P}$, which thereby becomes a language that is embedded in $\mathcal{L}$, see Fig. 4.

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

Figure 4. A $\mathcal{P}$ expression (XPath) embedded in an $\mathcal{L}$ expression (Java). From $\mathcal{L}$ perspective, $\mathcal{P}$ expressions are nothing but character strings.

It is good practice to delegate tasks to libraries, but if the task is PM it may be a bad idea. Some of the problems with an embedded $\mathcal{P}$ language are:
- There are no compiler warnings or debugging aids if there are errors in embedded $\mathcal{P}$ expressions. This is very unfavourable, because patterns are often the real workhorses in LT, the most difficult to get right and most in need of debug facilities.
- From $\mathcal{L}$ perspective, $\mathcal{P}$ expressions are more or less without structure. See Fig. 4. This can result in clumsy and unsafe code.
- **From $\mathcal{P}$ perspective, $\mathcal{L}$ expressions do not even exist. $\mathcal{L}$ expressions cannot be embedded in $\mathcal{P}$ expressions.**

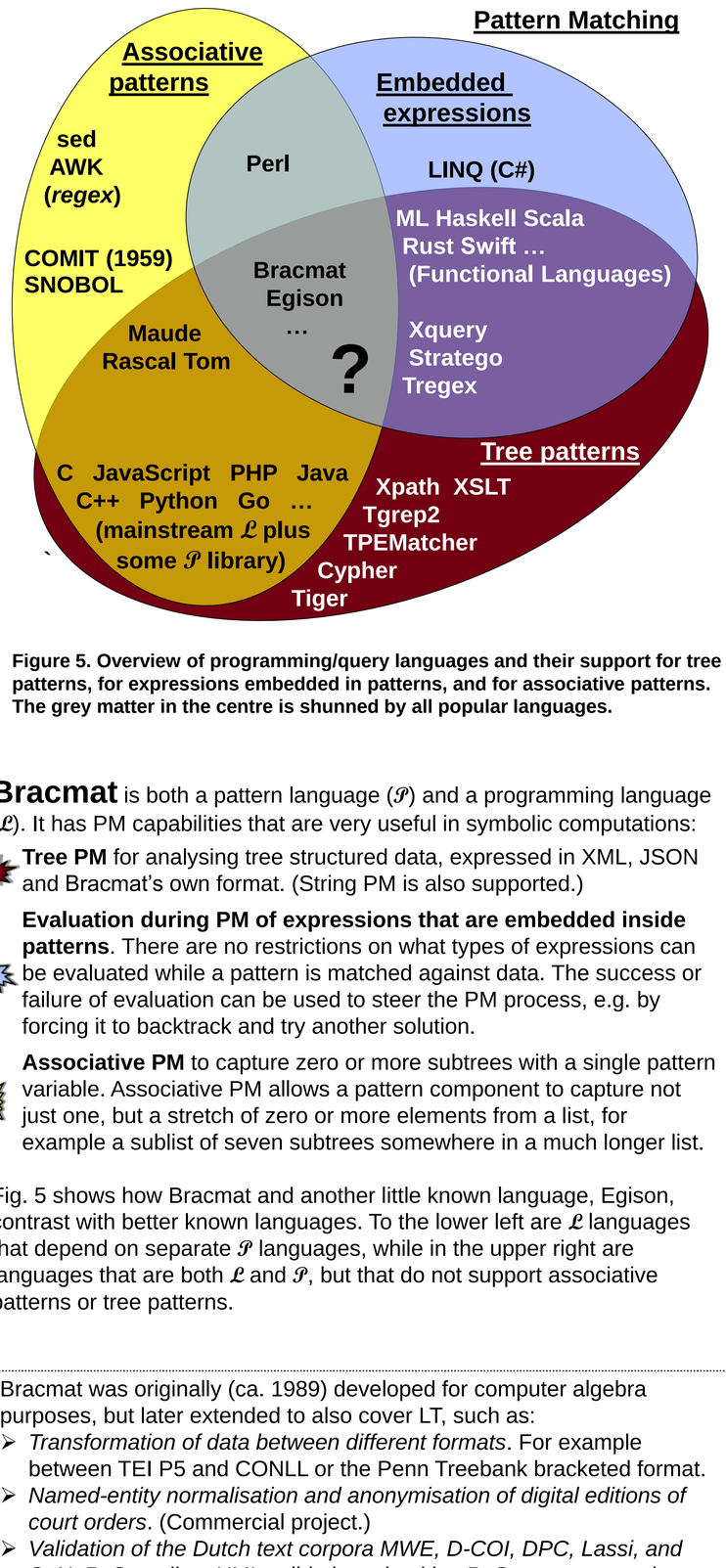

Figure 5. Overview of programming/query languages and their support for tree patterns, for expressions embedded in patterns, and for associative patterns. The grey matter in the centre is shunned by all popular languages.

Bracmat is both a pattern language ($\mathcal{P}$) and a programming language ($\mathcal{L}$). It has PM capabilities that are very useful in symbolic computations:

Tree PM for analysing tree structured data, expressed in XML, JSON and Bracmat's own format. (String PM is also supported.)

Evaluation during PM of expressions that are embedded inside patterns. There are no restrictions on what types of expressions can be evaluated while a pattern is matched against data. The success or failure of evaluation can be used to steer the PM process, e.g. by forcing it to backtrack and try another solution.

Associative PM to capture zero or more subtrees with a single pattern variable. Associative PM 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.

Fig. 5 shows how Bracmat and another little known language, Egison, contrast with better known languages. To the lower left are $\mathcal{L}$ languages that depend on separate $\mathcal{P}$ languages, while in the upper right are languages that are both $\mathcal{L}$ and $\mathcal{P}$, but that do not support associative patterns or tree patterns.

Bracmat was originally (ca. 1989) developed for computer algebra purposes, but later extended to also cover LT, such as:
- *Transformation of data between different formats*. For example between TEI P5 and CONLL or the Penn Treebank bracketed format.
- *Named-entity normalisation and anonymisation of digital editions of court orders*. (Commercial project.)
- *Validation of the Dutch text corpora MWE, D-COI, DPC, Lassi, and SoNaR*. Sampling, XML validation, checking PoS-tag usage and more.
- *Multimodal communication in a virtual world*. Keeping track of the dialogue between user and virtual agent and generating responses.

Example: Simplification of tool profiles

While most other programming languages are confined to a few colours in Fig. 5, Bracmat is at home in all corners, and never forces us to code ugly hacks like the one illustrated in Fig. 4. That could be reason enough to hail Bracmat, but the burning question is of course whether also the grey matter in the centre is useful, since so few languages are familiar with it. Therefore we go through code taken from the WMS that demonstrates the utility of an expression that is embedded in an associative tree pattern.

Suppose that the set of registered tool input/output profiles, due to a recent update, now has three incarnations of the same tool, a 'Brill' Part of Speech tagger. (See Fig. 6. For clarity, some details are replaced by …)

Here is a translation of the code in Fig. 6 to English:

"All three incarnations of the Brill tagger require segments and tokens as input and produce PoS tags. The incarnation for the English language outputs Penn Treebank tags, while the two Danish incarnations output Parole tags. The Danish incarnations can, optionally, take NER annotations (recognized Named Entities) as input. The tagger can handle unstructured, 'flat', text or TEI P5 text."

```
tools =
    ...
  + ( Brill-tagger
    .   (type,(segments*tokens.PoS^PennTree))
        (format,(TEIP5.TEIP5)+(flat.flat))
        (language,(en.en))
    )
  + ...
  + ( Brill-tagger
    .   (type,(segments*tokens NER.PoS^Parole))
        (format,(TEIP5.TEIP5))
        (language,(da.da))
    )
  + ...
  + ( Brill-tagger
    .   (type,(segments*tokens NER.PoS^Parole))
        (format,(flat.flat))
        (language,(da.da))
    )
  + ...
```

Figure 6. Excerpt of registered tool metadata, before simplification.

On closer inspection, we can see that the second and third incarnation can be combined, since those incarnations are different in only one feature: format. Combination of incarnations makes the visual display more attractive and speeds up the computation of viable workflows.

The number of incarnations in Fig. 6 can be reduced by a general algorithm that requires just a few lines of Bracmat code, see Figs. 7-8.

```
  !tools
: (   ?A
    + (?ToolName.?Features1)
    + ?M
    + ( !ToolName
      .   ?AA
          ( (?FeatureName,?Values2) ?ZZ
          &   !Features1
            : !AA (!FeatureName,?Values1) !ZZ
          )
      )
    + ?Z
  )
```

Associative tree pattern with embedded expression

Embedded expression

Associative tree pattern

Figure 7. Program that finds tool incarnations that can be combined.

Fig. 7 must be read as follows: "Find two incarnations of a tool that are equal except for one feature. Bind all other tool incarnations to the pattern variables `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`."

```
    !A
  + !M
  + ( !ToolName
    .   !AA
        (!FeatureName,!Values1+!Values2)
        !ZZ
    )
  + !Z
: ?tools
```

Tree pattern

Figure 8. Program that rewrites the list of tools by reducing the number of incarnations.

The code in Fig. 8 merely constructs a new list of tool incarnations and assigns the result of the construction, shown in Fig. 9, to the variable `tools`, thereby replacing the original value (Fig. 6).

This shows that expressions embedded in associative tree patterns can be concise and useful.

```
tools =
    ...
  + ( Brill-tagger
    .   (type,(segments*tokens.PoS^PennTree))
        (format,(TEIP5.TEIP5)+(flat.flat))
        (language,(en.en))
    )
  + ...
  + ( Brill-tagger
    .   (type,(segments*tokens NER.PoS^Parole))
        (format,(TEIP5.TEIP5)+(flat.flat))
        (language,(da.da))
    )
  + ...
```

Figure 9. Excerpt of registered tool metadata, after simplification.

Availability

The WMS and Bracmat can be downloaded as source code from GitHub:
https://github.com/kuhumcst/DK-ClarinTools
https://github.com/BartJongejan/Bracmat (includes technical paper for LT)
For inspiration: hundreds of tasks at rosetta.org are solved with Bracmat.

The WMS can be accessed from the CLARIN Language Resource Switchboard: http://weblicht.sfs.uni-tuebingen.de/clrs/.

contact: Bart Jongejan
Department of Nordic Research
Njalsgade 136
2300 Copenhagen S
Denmark
e-mail: bartj@hum.ku.dk