A tour through Bracmat

Bart Jongejan[*] BARTJ@HUM.KU.DK

[*] *NFI-University of Copenhagen, Njalsgade 140-142, Copenhagen, Denmark*

Abstract

We introduce and explain the features of the programming language Bracmat that are most useful for computational linguists.

Keywords: semistructured data; associative pattern matching; embedded expression evaluation; expression simplification; homoiconic programming language

1. Introduction

Linguistic research data is often semi-structured, for example the bracketed expression (1) in the Penn Treebank[1], which is the result of parsing and annotating the sentence in string (2). We will use this tree structure throughout this document to illustrate what Bracmat can do with it.

Penn treebank data as bracketed expression (1)

```
( (S (CC So)
     (NP-SBJ (PRP he) )
     (VP
       (VP (VBD sought)
         (NP (PRP her) )
         (PRT (RP out) ))
       (, ,)
       (CC and)
       (VP (VBD spoke)
         (PP (TO to)
           (NP (PRP her) )))
       (, ,)
       (CC and)
       (VP (VBD thought)
         (PP (IN of)
           (NP
             (NP (PRP$ his) (NN hand) )
             (PP (IN in)
               (NP (PRP$ her) (NN hair) ))))))
     (. .) ))
```

"So he sought her out, and spoke to her, and thought of his hand in her hair." (2)

1. the 15th tree in file cp28.mrg in the Treebank-3 material of the Penn Treebank project

2. Bracmat expressions

All Bracmat expressions are binary tree structures. Terminal nodes are strings, and they are normally written without surrounding quotes. Non-terminal nodes are operators, such as for adding or concatenating the node's left hand side (LHS) and right hand side (RHS).

Penn treebank data as Bracmat expression

```
( S
.   (CC.So)
    (NP-SBJ.PRP.he)
    ( VP
    .   ( VP
        . (VBD.sought) (NP.PRP.her) (PRT.RP.out)
        )
        (",",",")
        (CC.and)
        (VP.(VBD.spoke) (PP.(TO.to) (NP.PRP.her)))
        (",",",")
        (CC.and)
        ( VP
        .   (VBD.thought)
            ( PP
            .   (IN.of)
                ( NP
                .   (NP.("PRP$".his) (NN.hand))
                    ( PP
                    .   (IN.in)
                        (NP.("PRP$".her) (NN.hair))
                    )
                )
            )
        )
    )
    (".",".",".")
)
```

(3)

All nodes can be prefixed with one or more prefixes. Quotes can be put around a string, but they are only required if the string contains one or more characters that otherwise would be interpreted as operator or prefix. Parentheses are used to overrule the normal precedence of operators or when an operator needs to be prefixed.

Expression (3) is a translation of expression (1) to Bracmat format. Many non-terminal nodes in tree structure (3) contain dot-operators, and those that do not contain dot-operators contain white space operators. A Bracmat expression headed by a dot-operator is quite similar to a dotted pair in Lisp, but apart from that similarity Bracmat and Lisp are very different. The dot operator is an inert 'cement' in many fixed data structures. The white space operator is used in lists and has some more interesting properties that will be discussed later in more detail.

In the sequel we will use expression (3) in several code examples, but to save place, we store the expression in a file called 'cp2815' and then, instead of writing the full expression in each example, we use the expression `get$cp2815` that reads file 'cp2815' and then evaluates to expression (3).

3. Expression simplification

The basic tenet of Bracmat is that expressions evolve to normalized, maximally simplified forms. All the time keeping intermediary results simplified is a good strategy to avoid that computations in later steps become inextricably complicated and perhaps even impossible to perform. Normalization of expressions is to some degree achieved by Bracmat's expression evaluator, without requiring programming. This is best illustrated with algebraic expressions. The expression evaluator sorts and merges terms and factors[2] and applies basic simplification rules to pairs of juxtaposed

2. This is Associative Commutative normalization.

elements:

$$(b+a)+(a+2*b) \underset{sort}{\Rightarrow} (a+b)+(a+2*b) \underset{merge}{\Rightarrow} a+(a+(b+2*b)) \underset{add}{\Rightarrow} 2*a+3*b\ . \tag{4}$$

In contrast to addition and multiplication, concatenation of two lists (appending the second list after the last element of the first list) is not commutative. Lists of concatenated elements or trees are therefore not merged, but only converted to left-child right-sibling binary trees:

$$(The\ sun)\ \ (is\ red) \underset{concatenate}{\Rightarrow} The\ \ (sun\ \ (is\ red)) \Rightarrow The\ sun\ is\ red\ . \tag{5}$$

In the last step in (5) nothings happens to the data; we merely drop the parentheses because they are not needed.

If the algebraic operations were the only operations Bracmat mastered, the expression evaluator would not be able to simplify an expression if that required steps where elements had to be combined that are not juxtaposed, such as in this case:

$$\frac{b+x+by+xy}{1+y} \underset{?}{\Rightarrow} b+x \tag{6}$$

4. Programmability

A general algorithm that can simplify expression (6) is hard to implement, test and debug if directly written in C, the language in which Bracmat is implemented, or in any other mainstream programming language. Therefore Bracmat is a programming language in its own right, with high level language constructs like assignment, conditional branching and not to forget a pattern matching facility. With the assistance of these programming language constructs the expression evaluator is able to also simplify expression (6).

Programmability was achieved without changing the grammar of Bracmat. All programming constructs are implemented using strings, binary operators and prefixes. Since no distinction is made between data and program instructions, Bracmat is a homoiconic programming language.

The illusion of expression evolution is created by disassembling existing expressions and composing 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:

expression evolution

```
1 ( my cute dogs run:?D dogs %@?V
2 & do !D cats !V too "?"
3 )
```

(7)

⇒ `do my cute cats run too ?`

The binary operator `:` is the pattern matching operator. Its LHS is the subject of the operation, a list of four symbols (string tokens) linked by three space (concatenation) operators[3]. The RHS is the pattern `?D dogs %@?V` that has three components linked with space operators. In all examples in this text the pattern expressions are coloured **red** or **orange**.

Pattern components consisting of a string without prefixes (`dogs`) match only that string. Pattern components with a `?`-prefix (`?D` and `%@?V`) can capture values and are therefore pattern variables. Each time there is a match, the captured value is bound to the pattern variable.

A pattern component with `%` prefix can match one or more elements, but not a neutral element. A pattern component prefixed with a `@` only matches a neutral element or single terminal node (string). Together they ensure that `%@?V` matches exactly one string.

The operator `&` means 'and then'. It first evaluates its LHS and then, if the evaluation was successful, its RHS. Like the 'or else' operator | it has short-circuit semantics.

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 of those variables.

3. Any number of white space characters between two subexpressions corresponds to one space operator. The white spaces in `x + y` are not space operators, because a solitary `+` is not a subexpression.

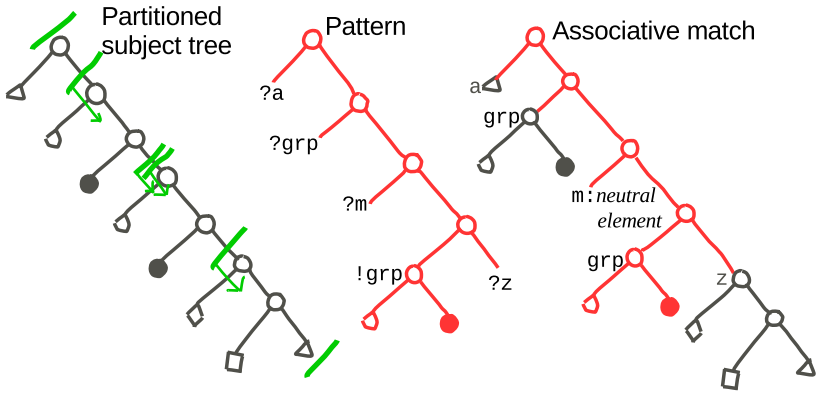

Figure 1: Associative pattern matching. The subject is a normalized expression with the same associative operator in all seven nodes that constitute the 'spine'. The (non-linear) pattern creates a partition of the subject that isolates two groups with identical labels. The pattern variables `a` and `z` capture the head and the tail of the subject, while the variable `m` captures the neutral element of the associative operator.

5. Associative pattern matching

Associative pattern matching against sums, products, and also lists of concatenated subtrees, enables Bracmat to reason about algebraic expressions like (6) or semi-structured representations of natural language expressions like (3). For associative pattern matching to work best, we need elements that can be inserted anywhere in a sum, a product, or a list. The operators for addition, multiplication and concatenation have such neutral elements: 0, 1 and "" (zero length string), respectively. These neutral elements come and go as needed. Neutral elements function as stand-ins for 'no elements' in associative pattern matching (see the pattern variable `m` in Fig. (1)), while they dissolve in the presence of a non-neutral neighbouring element during expression simplification.

In Fig. (1) the subject tree and the pattern seem to have incompatible structures. While the subject tree has a straight spine of associative operators, the pattern has an associative operator in a subtree `!grp` as the LHS of the rightmost non-terminal node. The pattern matcher takes care of this mismatch during associative pattern matching. Because the operators in the subject tree are associative, the pattern matcher ignores the differences between the actual structures of the pattern and the subject tree. The inverse process is also possible. The pattern variable `?grp` matches two consecutive children in the subject tree. When asked to evaluate `!grp`, Bracmat creates a new node with the two matched nodes as children. This works for any number of consecutive nodes matched by the pattern variable, but there is a catch. Evaluating a variable that only matches part of a list, but that does not include the tail of the list, has a cost in both time and memory that is proportional to the number of captured consecutive nodes, because new nodes must be created for the full length of the copied part of the spine. Matching the tail of a list, on the other hand, does not involve the overhead of creating new nodes and has a constant, low cost.

The shown partition of the subject tree in Fig. (1) is only one out of many possible partitions. To force the pattern matcher to choose precisely this partition and not one of the other partitions, some constraint must be added to the pattern that makes it fail with other partitions. In this example, the constraint is that the same variable occurs twice, first receiving `?grp` and later producing (`!grp`). One of the partitions that satisfies this constraint is the shown partition.

Fig. (1) is an illustration of a non-linear associative pattern. An associative pattern is non-linear if a pattern variable is capturing and later in the same pattern producing. The Bracmat expression (8) shows another example. The expression first looks for a constituent of the sentence in example (3) that has two (or more) children `Child1` and `Child2` with the same label, and then

creates a little report. The pattern component `!Lbl` evaluates to the latest value captured by the `?Lbl` pattern component and is what makes the pattern non-linear.

A solitary `?` (an empty string prefixed with `?`) plays the role of a wild card, in this example matching an empty string or one or more dotted pairs.

non-linear associative pattern

```
1 (   get$cp2815
2   : ( S
3     .   ?
4         (?ParLbl.? (?Lbl.?Child1) ? (!Lbl.?Child2) ?)
5         ?
6     )
7 & Found !Child1 and !Child2 both labeled !Lbl in !ParLbl constituent
8 )
```

(8)

⇒ `Found (VBD.sought) (NP.PRP.her) (PRT.RP.out) and (VBD.spoke) (PP.(TO.to) (NP.PRP.her)) both labeled VP in VP constituent`

6. Procedures and other unevaluated code

When programmability was introduced, a need arose to keep Bracmat code in an unevaluated state, so that the code repeatedly could be copied and evaluated, for example in an iterative process. To that end the `=` operator was introduced. This operator only evaluates its LHS. If the LHS of the `=` operator is a non-zero length string, the RHS of the `=` operator is bound to that string, which thereby becomes a variable. In this way a procedure that has to be run several times can be bound to a variable and each time the variable is evaluated, the procedure is run anew. Example (9) shows how one can invert a list by repeatedly removing the first element from a list and prepending the removed element to a new, initiallly empty, list, using a procedure bound to the variable `loop`. This procedure works as follows. The `%` prefix forces the pattern matcher to let the subject of the pattern component `%?first` grow to (at least) one non-neutral element. The pattern matching expression `!oldList:%?first ?oldList` therefore removes a single element from `oldList`. When, after some iterations, `oldList` has been shortened until only a neutral element is left, the pattern matching operation fails, causing the loop expression to fail, causing the LHS of the `|` operator to fail. Finally the expression `!newList` is evaluated on line 9.

non-linear associative pattern

```
1 ( (VBD.sought) (NP.PRP.her) (PRT.RP.out):?oldList
2 & :?newList
3 & ( loop
4   =   !oldList:%?first ?oldList
5     & !first !newList:?newList
6     & !loop
7   )
8 & (!loop|!newList)
9 )
```

(9)

⇒ `(PRT.RP.out) (NP.PRP.her) (VBD.sought)`

There is a second way to create unevaluated code and keep it unevaluated: the macro expansion mechanism. We introduce two binary operators, the `'` (apostrophe) and the `$` (dollar sign). Expression (10) illustrates macro expansion.

The `'` operator searches its RHS for `$` operators that have an empty string as LHS. Each such expression is replaced by the value of the variable that is on the RHS of the `$` operator, so `$Lbl` is replaced by `VP` and `$grpPat`[4] is replaced by the unevaluated expression `(",".?)  (CC,?)`. Finally, the `'` operator itself is replaced by the aforementioned `=` operator. Using macro expansion, it is possible to build unevaluated code in small steps, in this case a pattern expression.

4. The `()` preceding `$grpPat` is needed to avoid that `($Lbl.?)` is taken as the LHS of the `$` operator.

All expressions in this text where macro expansion takes place, are coloured **blue**.

macro expansion

```
( VP:?Lbl
& (grpPat=(",".?) (CC,?))
& '((($Lbl.?) ()$grpPat)
)
```

(10)

⇒ =(VP.?) (",".?) (CC,?)

7. Success, failure, control structures, the 'cut' prefix, and negation

The evaluation of a Bracmat expression either fails or succeeds. The most common and regular cause of failure is a pattern that does not match a subject, but expressions can also fail for other reasons, such as a non-accessible file or an uninstantiated variable. 'Success' and 'failure' are analogous to 'true' and 'false' in many other programming languages, and Bracmat has control structures analogous to the *if ... then ... else* programming construct that react to the success or failure of expressions and patterns: conjunctions and disjunctions.

The operator `|` can be used to create a disjunction of patterns, the same operator as used for disjunction elsewhere. The operator `:` can be used to create a conjunction of patterns, the same operator as used for pattern matching. Both disjunction and conjunction of patterns are short-circuit control structures.

Example (11) shows disjunction (`|`) and conjunction (`:` resp. `&`) in pattern context as well as outside pattern context. The pattern searches for a sentence that starts with a constituent labeled `CC`, `PP` or `ADVP`. If that part of the pattern succeeds, the found constituent is matched by the variable `?constituent`. If the pattern match operation on the whole pattern succeeds, `!constituent` is evaluated. If the match operation fails (or the file cp2815 cannot be read) the words **`not found`** are returned.

logical operators

```
(     get$cp2815
    :   ?
        ( S
        .   ( (CC|PP|ADVP.?)
            : ?constituent
            )
            ?
        )
        ?
  & !constituent
| not found
)
```

(11)

⇒ CC.So

Backtracking pattern matching operations can take a long time before terminating. In many situations Bracmat can apply heuristics to restrict the search space. It knows, for example, that matching a leaf pattern with any subject that is not a leaf, never will succeed. In some cases the programmer needs to explicitly tell Bracmat to restrict the search space, if time and memory are issues. To prevent unnecessary backtracking we can prefix a pattern component with a `` ` `` (backquote or grave accent): it tells the pattern matcher to fail at once if the current match failed. We use this prefix in situations where we allow at most one subtree of a list of trees to be matched.

Sometimes we want to change success to failure and vice versa, for example if we want to make sure that a subject does not match a certain pattern, or to force Bracmat to backtrack to an earlier decision point and then find alternative solutions. Negation of patterns and other expressions is achieved with the `~` prefix. The following pattern (12) contains two negations and succeeds if there are no values in a list of (*name.value*) pairs that are non-empty strings. The first `~` negates the

complex pattern inside the parentheses and the second ~ negates the empty string.

negation of complex pattern and leaf pattern

```
1     (a.) (b.) (c.blah) (d.): ~(? (?.~) ?)
2   & "All values are empty strings"
3   | "Not all values are empty strings"
```

(12)

⇒ **Not all values are empty strings**

The same result is obtained by negating the match expression instead of the pattern:

negation of expression and of leaf pattern

```
1     ~((a.) (b.) (c.blah) (d.):? (?.~) ?)
2   & "All values are empty strings"
3   | "Not all values are empty strings"
```

(13)

⇒ **Not all values are empty strings**

8. No $\mathcal{L}$ - $\mathcal{P}$ split

Often, pattern matching in a programming language $\mathcal{L}$ is implemented in a library, introducing an embedded pattern language $\mathcal{P}$ wherein patterns must be expressed. This split of responsibilities is asymmetric. While $\mathcal{L}$ can invoke $\mathcal{P}$ expressions, $\mathcal{P}$ cannot invoke $\mathcal{L}$ expressions, thus limiting the expressiveness of patterns.

Bracmat gives full access to the language inside patterns. The `&` operator can be used to decorate a pattern component with an expression that is evaluated after each successful match of that component. The following example finds and accumulates the top level labels in a sentence:

pattern with embedded expression

```
1  ( :?Labels
2  & (   get$cp2815
3      : ( S
4        .   ?
5            (   ?Label
6              & !Labels !Label:?Labels
7              & ~
8            . ?
9            )
10           ?
11       )
12   | !Labels
13   )
14 )
```

(14)

⇒ `CC NP-SBJ VP "."`

The pattern starting on line 3 and ending on line 11 is interrupted for an intermezzo on lines 6 and 7. The pattern, `(S.? (?Label.?) ?)`, matches if its subject is a dotted pair with as its LHS an `S` symbol and as its RHS a list with at least one element that, again, is a dotted pair. The LHS of the dot operator is captured by the variable `Label`.

The `?Label` pattern component is decorated with the expression

embedded expression

```
1 !Labels !Label:?Labels & ~
```

(15)

Expression (15), which contains a complete pattern matching expression, adds the found label to the accumulator variable `Labels` that was initialized with an empty string at the beginning of the program (`:?Labels`) on line 1. It then fails due to the always failing expression `~`. That forces the pattern matcher to backtrack and try to find another label in the list. Finally, when no more labels can be found, control is passed to the right hand side of the `|` operator, and the accumulated value bound to `Labels` is returned. Patterns in expressions like (15), that occur inside other pattern matching expressions, are coloured **orange** in this text.

9. Functions

No programming language is complete without a possibility to define and call functions. Bracmat has user-definable and built-in functions. We have already seen an example of the latter: the `get` function. In this text we use three built-in functions: `get` that reads Bracmat expressions from file, console or string, `lst` that writes a listing of an expression to a file, console or string, and `whl` that implements a loop structure.

The `$` operator applies the function mentioned in its LHS to the expression on its RHS. Another operator that does the same is the `'` (apostrophe) operator. The difference between these two operators is that the `$` operator evaluates its RHS before passing it to the function, while the `'` operator passes its RHS to the function without first evaluating it.

User-defined functions are written as Bracmat expressions. The syntax is

```
( functionName = local variables . function body )
```
(16)

The `=` has a low priority, so everything right of the `=` operator, until the closing parenthesis, is unevaluated code. Local variables are declared in the LHS of the `.` (dot) operator, while the RHS is the function body that has to be evaluated when the function is called. The result of the evaluation of the function body becomes the value returned from the function.

The actual argument expression to a function is always bound to a local variable called `!arg` that needs no explicit declaration. Pattern matching can be used to retrieve parts of the argument expression.

function definition and use

```
( ( words
  =   Child Sibling
    .   !arg:@
      |   !arg:(?.?Child) ?Sibling
        & words$!Child words$!Sibling
  )
& words$(get$cp2815)
)
```
(17)

```
⇒ So he sought her out , and spoke to her , and thought of his hand
   in her hair .
```

Expression (17) defines a function that prints out the unannotated sentence enclosed in Penn Treebank sentence tree (3).

Function calls appearing in patterns are evaluated during pattern matching. If the pattern matcher is forced to backtrack a pattern function may be called several times. Pattern functions have a second argument: that part of the subject that the pattern function is trying to match. This second argument is bound to the implicitly declared local variable `!sjt`. Pattern functions are useful for creating recursive patterns that require a fresh set of local variables at every invocation of the recursive pattern.

10. String pattern matching

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:

PM on structured data vs string PM

```
t h o u g t:? o u g h ?     tree PM
@(thought:? o u g h ?)      string PM
@(thought:? ough ?)         string PM, more efficient
```
(18)

The `@` (‘atom’) prefix is attached to the `:` operator, which is the root node of the parenthesized expression. It tells Bracmat to use string pattern matching instead of the default tree pattern matching.

It is also possible to combine tree pattern matching and string pattern matching. Example (19) checks whether the subject (from sw2305.mrg in Penn Treebank) contains three words that begin with the same letter.

Mixing PM on structured data and string PM

```
(   (NN.soy) (NN.sauce) (CC.and) (NN.sesame)
  : ? (?.@(?w1:%@?c ?)) ? (?.@(?w2:!c ?)) ? (?.@(?w3:!c ?)) ?
& !w1 !w2 !w3
)
```

(19)

⇒ soy sauce sesame

11. Position pattern

The prefix `[` creates a pattern component that is sensitive to the current position of the neighbouring pattern components in the subject of the pattern. The pattern `[?p` counts the number of list elements (concatenation, sum or product) until the beginning of the list and puts that number in the variable `p`. The pattern `[12`, on the other hand, only succeeds if there are exactly 12 elements to the beginning of the list. Negative numbers can be used to count towards the end of the list, so `[-3` only succeeds if there are 3 members before the list ends. Instead of numbers we can also use variables. If `q=4`, then `[!q` only succeeds if there are four elements until the start of the list. Prefixes `<`, `>` and `~` can be used together with a number or number producing variable to indicate that the number of elements to the beginning of the list must be less than, greater than or unequal to that number.

Example (20) shows how to measure the length of a list using the position prefix.

measure the length of a list

```
( get$cp2815:? (S.? (VP.? [?length) ?) ?
& !length
)
```

(20)

⇒ 7

12. Reading and Writing

The easiest and safest way to read and write files is by use of the `get`, `put` and `lst` functions.

We have already seen the `get` functions in use in several examples. In all those examples the input file contained one or more Bracmat expressions. By adding optional parameters we can instruct the `get` function to read other types of input files as well. Currently, `get` can read marked-up text (SGML, XML and HTML) and JSON data as semistructured data, and text as a single string. Parsing marked-up text and JSON is directly implemented in C and therefore very fast. Once read, the data is available like any other Bracmat structure and can be parsed, transformed, and written. In the internal representation of marked up text XML entity references and HTML entities are converted to characters. By adding the option `TRM` all strings only containing whitespace characters are converted to a single whitespace. Numeric data in JSON data is converted to rational numbers[5]

Example XML file ``books.xml''

```
<?xml version="1.0" encoding="utf-8"?>
<!-- XML example -->
<books class="abc">
    <book id="1" type="Action">Ian</book>
    <book id="2" type="Humor">Ali</book>
</books>
```

(21)

5. Bracmat has no floating point data type.

Reading marked up text

```
( get$("books.xml",X,ML,TRM):?xml
& nestML$!xml:?xml
& lst$xml
)
```

⇒

```
(xml=
  ("?"."xml version=\"1.0\" encoding=\"utf-8\"")
  " "
  ("!--"." XML example ")
  " "
  ( books
  .   (class.abc)
    ,   " "
        (book.(id.1) (type.Action),Ian)
        " "
        (book.(id.2) (type.Humor),Ali)
        " "
  )
  " "
);
```

(22)

Both the `put` and the `lst` function can write data to a file. The difference is that `lst` writes data that is meant to be read again, such as program listings, tables and other data that are Bracmat expressions, while `put` is meant to write strings without the quotes and escape characters that `lst` has to utilize.

While `get` reads a whole file in one piece, `put` and `lst` can be instructed to append output to an already existing file. The option `NEW` tells Bracmat to write a new file, overwriting an existing file, if needed, while `APP` instructs Bracmat to append to an existing file, or to create a new file if the file does not exist yet.

Convert Bracmat data to marked up text and write to file

```
put$(toML$!xml,"newbooks.xml",NEW)
```

⇒

(23)

```
<?xml version="1.0" encoding="utf-8"?> <!-- XML example -->
 <books class="abc"> <book id="1" type="Action">Ian</book>
 <book id="2" type="Humor">Ali</book> </books>
```

Handling JSON is similar to handling marked up text. Instead of the options `X,ML` (or `HT,ML`) one uses `JSN`. For converting Bracmat data to a JSON formatted string one can use the `jsn` function.

Use the option `STR` to read a text file into a single string.

Read text file

```
get$("MobyDick.txt",STR):?moby-dick
```

(24)

There is also a low level function `fil` for opening, reading, writing and closing a file. This function can handle binary files, i.e. files that can contain null bytes.

For further details about reading and writing in Bracmat we refer to the help file.

13. Example: Detect Repeating Grammatical Groups

In this section we discuss a query that is hard or impossible to implement in most query languages, and that would require many lines of code in a programming language. The Bracmat inmplementation is concise and, if you understand each detail, easy to understand.

The query has to find constituents with two groups of child constituents with the same sequence of labels. Within a constituent no more than two children are allowed to not be included in the repeating group. For this query we need associative pattern matching as well as expression evaluation during pattern matching.

Query 5: Detect repeating grammatical groups

```
1  ( ( recursivePattern
2    =   ?Tag
3      .       ?a
4              ( (?.?) (?.?) ?:?grp (?Lbl.?)
5              & '($Lbl.?):(=?grpPat)
6              &   whl
7                ' ( !grp:?grp (?Lbl.?)
8                  &   '((($Lbl.?) ()$grpPat)
9                      : (=?grpPat)
10                 )
11             )
12             ?m
13             !grpPat                                                    (25)
14             (?z&!a !m !z:? [<3)
15         : ( ?Phrase
16           & (!grpPat.!Tag.!Phrase) !Acc:?Acc
17           & ~
18           )
19       | ? ((?.?):!recursivePattern) ?
20   )
21 & :?Acc
22 & ( get$PennTrees:?done ((?.?sentenceTree):!recursivePattern) ?todo
23   | lst$(Acc,"report.txt",NEW)
24   )
25 )
```

Expression (25) is the complete Bracmat program for detecting and reporting repeating groups. The expression `recursivePattern` does almost all of the work. It contains a disjunction of two patterns. Either the pattern successfully matches the first condition (lines 3-18), or it recurses deeper into the tree structure (line 19). Because the first condition is never satisfied, the pattern traverses a complete sentence tree, but finally fails. When that happens the next sentence is tried (line 22). When all sentences have been visited, control passes to line 23, where a report is written to a file.

Even though `recursivePattern` is programmed to fail, it finds and accumulates all instances of the sought-for repeating grammatical groups that are present in the input. Finding those instances is done in lines 3-14 and remembering them for the report is done in line 15-18. Line 4 is a subpattern that matches a group of two or more constituents and assigns that group to the variable `grp`, except the last constituent, which is assigned to the variable `Lbl`. In lines 6-10 the labels of the constituents in `grp` are transferred to a pattern variable `grpPat`, which is initialized using the value of `Lbl`. The pattern variable `grpPat` is used a few lines lower, on line 13. On lines 3, 12 and 14 we see pattern variables `a`, `m` and `z` that grab the remaining elements.

The expression `recursivePattern` is used on line 22 as a pattern component in another pattern. That pattern has three top-level components, one for the trees already done, one for the trees to come and, between those, one for the current tree. The latter selects a single sentence tree with the pattern `(?.?sentenceTree)` and then applies `recursivePattern` to that sentence tree, but since `recursivePattern` is programmed to fail, Bracmat backtracks and then selects the next available sentence tree. In that way, every sentence tree is matched against the recursive pattern `recursivePattern`.

14. Pitfall

Users of Bracmat should beware of unnecessary backtracking, which can increase processing time to an unacceptable height. An example of this is the following:

long winding pattern matching operation

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

(26)

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 very 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 against one `x`, two `x`s, three `x`s, ..., forced by backtracking when either `!p` fails (every second time) or the subpattern `(x|y)` fails. This process repeats at each level of recursive invocation of the pattern `!p`.

To write an efficient pattern, the programmer should economize 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:

optimized pattern matching operation

```
1   ( q = | x ?Betweeners (x|y) & !Betweeners:!q )
2 & x x x x x x x y x x : !q
```

(27)

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

15. Further reading

The Bracmat repository at GitHub[6] contains a help file with many more details of the Bracmat language.

For many examples see the programming chrestomathy of Rosetta Code[7]. This site presents solutions for the same task in many programming languages. There are at the time of writing 280 Bracmat examples in Rosetta Code.

6. https://github.com/BartJongejan/Bracmat
7. http://rosettacode.org/wiki/Rosetta_Code