Bracmat

Bart Jongejan

July 11, 2020

Contents

1 Introduction

Bracmat makes it a joy to find your way in oddly shaped data and manipulate it at your will. Bracmat occupies a niche were you find few, if any, other computer programming languages.

This chapter may give you an impression of what Bracmat can do for you.

1.1 What is Bracmat?

Bracmat is a computer programming language designed for analysis and manipulation of complex data, be it evaluation of algebraic expressions, parsing natural language, validation of HTML or automatic chaining multifacetted webservices into a workflow. In several projects, Bracmat has been combined with other programming languages to take care of high level activities that otherwise would be very time consuming to implement.

Bracmat was originally developed as a calculator for symbolic algebra. Like a table calculator, Bracmat ran a Read Eval Print Loop (REPL). It added, multiplied, took powers and logarithms and differentiated algebraic expressions.

Later additions transformed the calculator into a programming language. The subject matter, algebraic expressions, had often to be destructured and recombined in a different way, e.g. to find terms with like factors and recombining these into a group of new terms with a common factor. What was needed was advanced pattern matching with a notation that requires little more than your intuition to understand, because the algebraic transformations to be implemented were already hard enough to grasp. Envisioning that pattern matching also would be very useful in natural language related problems, the subject domain was extended to cover almost any kind of data.

True to its roots, Bracmat still has an important feature in common with calculators: as long as the input data are simple enough, the user does (and can) not specify how these data have to be processed. Calculators handle basic calculations in a predictable, unchangeable way, e.g. `4+7` will always result in `11`, and not in, say, `10` or `23-12`. That is because the manufacturer had good reasons to think that `11` is what the user expects and nothing else.

In the same way, Bracmat handles basic "calculations" with a much wider variety of data: rational numbers, symbols, words and collections thereof. For example, `a+b+a` becomes `2*a+b` and not, say, `x.a` or `b+2*a`. Again, Bracmat takes decisions that you can't easily circumvent. However, the more complex the data are, the better are the chances that not all results, although defensible, have an appearance that suits you. It is here that programming comes in: Bracmat leaves certain kinds of data unchanged, but opens the possibility to dissect data, to perform calculations on the parts, and to assemble an answer from the resulting subanswers.

Bracmat makes no distinction between data and instructions. All there is are expressions. Some expressions stay the same upon evaluation, other expressions change radically, and then there are expressions that contain some parts that are stable upon evaluation and other parts that change. The advantage of having the same syntax for data and instructions is not in the first place that you could deconstruct and recombine program code (generally not a good idea), but that you can embed code in a data structure that

adds more data upon evaluation. Such code can function as growing tips in list or tree structures, as variable fields in a standard letter, or as places of particular interest inside a pattern, to name some examples.

Another advantage of not having to distinguish between code and data is that reading a program and reading data from a file are no different. And saving a program is the same as saving data to a file. Which brings us to yet another enormous advantage: a Bracmat expression is virtually undecipherable if it doesn't obey the canonical layout rules, which are ... canonical. Whereas layout doesn't matter to Bracmat, it does to you, and therefore Bracmat, when asked to save an expression to file, imposes the canonical lay-out rules on the output, unless you insist on wanting to have it all in one line, no superfluous spaces allowed.

Bracmat expressions have a simple syntax: a mix of parentheses, binary operators, operands and prefixes preceding those operands or parentheses. When programmability was added to Bracmat, no new syntax was invented, but only a few handfuls of prefixes and binary operators dedicated to assignment, function calling, pattern matching and program flow, in addition to the operators for addition, multiplication, etc.

1.2 Why use Bracmat?

Bracmat is not a general purpose programming language, but a programming language that was conceived in the 1980's with a single aim: to make tedious algebraic manipulations easy and, most importantly, not prone to human errors. Now hang on.

Many problems can be solved by following steps similar to those taken when solving an algebraic problem:

- Find stuff
- Find the context of stuff
- Sort stuff
- Merge stuff
- Combine stuff
- Replace stuff
- Cope with labyrinthine problems with many culs-de-sac

1.3 User interaction with Bracmat

Bracmat offers a simple environment for input of both data and code. After the prompt `{?}` you can write your input. When you hit the return key, Bracmat evaluates your input and writes the result to the screen following a `{!}` sequence (unless there was no visible result). Under the result follows a line that tells whether the evaluation was successful (`S`) or not (`F`) (in rare cases you may see an `I`, which, for the time being, you may interpret as failure). In the same line the machine shows how much processor time it needed. Intermediary results may also appear on the screen.

When instructions are entered from the keyboard, the program waits until all of the conditions below are fulfilled:

- *return* (or *enter*) is the last key that has been pressed
- every opening parenthesis has a closing counterpart
- every opening brace (start of comment) has a closing counterpart
- every string has either two enclosing double quotes, or none.

You can freely have parentheses and double quotes in a comment and you can freely have parentheses and braces in a string, provided the string is enclosed in double quotes.

You can write a multiple-line instruction by putting the instruction inside an extra pair of parentheses. After each non-terminating *return*, Bracmat shows in the next line how many closing parentheses are needed for the completion of the instruction. If you are in the middle of a string or a comment when pressing *return*, the next line starts with `{str}` or `{com}`, respectively.

If you want to enter several instructions on the same line, you should write a semicolon `;` between the instructions. These instructions are executed in the same order. If you do not want to see the result of a calculation, you may write a semicolon after the last instruction. Instructions in a text file must be separated by semicolons.

You may freely surround binary operators with whitespace characters (e.g. space, tab, line feed). Take care not to put spaces between the characters that make up a fraction or negative number, `- 1234 / 5678` is not the same as `-1234/5678`.

Bracmat can open and read a text file and execute the instructions as they are read. After the last instruction has been executed, the file is closed. For example, the program that produces the text you are reading just now, is read from a file named `help` and immediately executed.

The instruction for reading a file `myprog` and executing the instructions therein is `get$myprog`. The `$` is a binary operator that initiates function evaluation. `get` is the function name of one of the few built-in functions.

The result of the last executed instruction in a file is written to the screen. For better control over screen output one may use the built-in function `put`, which writes from the current cursor position to the right, or the predefined (but changeable) function `out`, which writes an extra line feed after its argument has been written to the screen.

```
{?} put$(x*x);put$(y+y)
x^22*y{!} 2*y
```

Often you will need the result of the last evaluated instruction in the next instruction. You can use the exclamation mark `!` instead of re-entering the result. Example:

```
{?} 1+1
{!} 2
{?} !^!^!^!
{!} 65536
```

When you write programs in the Bracmat language you will normally use an external text editor. You can enter small programs directly at the Bracmat prompt `{?}`, but a small change in an instruction can only be done by re-entering the whole instruction. You may save instructions that are still held in memory, by the built-in `lst` function. This function takes a number of optional parameters that tell the system whether the instruction has to be written in a compact and barely readable form, or in a more pretty form, with lots of indentations. Comments are never written, as they are ignored at input time. As an example, you can write the definition of the predefined factorisation function `fct` to a file factorise by entering the following instruction:

```
{?} lst$(fct,factorise,NEW) { NEW: replace old file with same name }
{!} fct
```

At first sight, a Bracmat program doesn't look like programs written in other languages. This may even become a permanent impression. So here are some recommendations about programming habits:

- Experiment with the examples in this document.
- Use a modular and incremental style of programming: test every function before going on to the next one.
- Keep functions small, to begin with. Error messages point to functions, not to line numbers.
- Use meaningful names. There is no limit to their length and you may use the minus-sign as separator in multiple-word-names.
- Let Bracmat indent your code: regularly save your code, read it in Bracmat, save it from Bracmat using the lst function to the file you are editing and reread the file in your editor.
- Try to evade the "not" prefix `~` and the use of arrays.

1.4 Limitations

Oddly enough, Bracmat has no operators for subtraction and division. The reason is that these operations lack a very desirable property: associativity. Addition and multiplication are associative, because

```
a+(b+c)
```

is equivalent with

```
(a+b)+c
```

and can be written as

```
a+b+c
```

On the other hand, subtraction and division are not associative, as can be seen in this example

```
a-(b-c)
```

which is not equivalent with

```
(a-b)-c
```

because

```
a-(b-c) = a-b+c
```

while

```
(a-b)-c = a-b-c
```

So we see that an expression with binary minus operations and written without parentheses is a left descending tree: the topmost binary operator is the operator to the far right. And indeed, the first incarnation of Bracmat internally represented sums and products as left descending trees. However, we are used to see numerical factors to the left of non-numerical symbols. If sums and products are left-descending, adding `3*a*b*c+4*a*b*c` becomes unnecessary expensive

```
3*a*b*c+4*a*b*c = ((3*a)*b)*c+((4*a)*b)*c
```

The first expensive step is finding the numerical factors `3` and `4`, because they are several levels deep.

The second expensive step is that in the result

```
((7*a)*b)*c
```

all multiplication operators necessarily have to be new nodes, so to do this simple addition, 4 new nodes (one for the 7 and three for the multiplication operators) had to be created. Compare this with a right descending tree as the default internal representation for sums and products:

```
3*a*b*c+4*a*b*c = 3*(a*(b*c))+4*(a*(b*c))
```

The numerical factors are very close to the root of the tree and therefore easy to find. Moreover, for the result

```
7*(a*(b*c))
```

Bracmat only needs to create one node for the numerical factor `7` and one node for a single multiplication operator, because the subexpression

```
(a*(b*c))
```

can be shared with the original expression.

Fortunately, subtraction and division are unnecessary operations if you have negative numbers and the operations of multiplication and exponentiation, because

```
a - b
```

can be written as

```
a + -1 * b
```

and

```
a / b
```

can be written as

```
a * b ^ -1
```

A completely unrelated limitation is that in Bracmat, calculations always have to be exact. Number expressions for which no rational representation exists are not further evaluated. Bracmat knows how to handle the special symbols `i`, `e` and `pi`, but it offers no numerical representation for `e` and `pi`.

Examples:

```
{?} a+b+-1*a                { this is how you subtract in Bracmat }
{!} b
{?} 1+(1+i)*(1+-1*i)+-1 { the leading and trailing terms
                            force Bracmat to expand the product }
{!} 2
{?} 12345/54321 ^ 1/2     { the square root of 12345/54321 }
{!} 5^1/2*19^-1/2*823^1/2*953^-1/2
{?} x^(a+x\L2*100)
{!} 126765060022822940149670320537 6*x^a
{?} 5/2 \L 987654
{!} 15+5/2\L32363446272/3051757812
{?} y\D(x\D((x+y)^-2))
{!} 6*(x+y)^-4
```

Whereas Bracmat lacks floating point arithmetic, it can perform arithmetic operations with integer and rational numbers and happily adds and multiplies numbers with thousands of digits. Only available computer memory and address size impose a limit to the size of numerical operands, but before you decide to multiply two ten-million digit numbers you should realize that Bracmat is not optimised for number crunching on that scale.

Bracmat handles non-integer powers of positive rational numbers (square root, for example) provided that the number (if it is an integer) or the numerator and the denominator (if the number is a fraction) are less than `2^32` (or `2^64`, if Bracmat is compiled for a 64-bit platform).

There is a predefined function, `flt$`, that represents rational numbers in a scientific floating point notation, but Bracmat cannot do calculations with these floating point numbers, unless you write a function to convert them back to rational numbers.

1.5 How Bracmat evolved

Bracmat originated from a Basic program that was meant (and able) to do some algebraic calculations in General Relativity. This program could do the mathematical operations that Bracmat can: add, multiply, take powers and logarithms and differentiate. This calculator was not programmable, all program flow had to be done in Basic. It became clear that the program could only solve the simplest algebraic problems. The reason for this was its inability to recognise complex patterns in the subject expressions. All

pattern recognition had to be done in Basic and this was a very fault prone business. It would be nice to have an interpreter at hand that could interpret human readable production rules. That is exactly what Bracmat embodies. This program is written in ANSI C and developed on the fastest home computer that existed at that time (Acorn Archimedes). Although Bracmat is much faster than its predecessor, its main virtue lies in its programmability and the advanced pattern matching. The speed at which it processed formulae was not impressive, so Bracmat always needed a fast machine. Fortunately computers have become faster and faster, and nowadays Bracmat performs at a speed that is quite acceptable, even in real time systems with users expecting immediate responses.

Compared with other algebra systems, Bracmat has few built-in functions and even its set of mathematical operators is small. There are no operators for subtraction and division, for example. Nevertheless, Bracmat is general and flexible enough to solve even problems outside the field of computer algebra in an elegant way. This flexibility on the programming level is traded off against the inability to change the behaviour of the interpreter itself. There are no switches (toggles) that could influence, for example, the order of terms within a polynomial, or whether or not complex products are expanded, or the way backtracking is done. This choice was, of course, easier to implement, but it also has benefits for the user: the working of Bracmat programs is not obscured by deep side effects of switch settings. The only side effects that Bracmat allows are expression binding and change of focus in a multiple valued variable (array indexing, stacking). A later addition is support of objects, i.e. data structures that allow for partial updates. This introduced another kind of side effect.

One peculiar thing about the original, object-less Bracmat is the way in which it manages data:

1. Processes are not periodically interrupted for garbage collection.
2. Each piece of data has a reference counter. If the reference counter equals zero, the occupied memory is returned to the memory pool at once. If the reference counter is about to overflow, a fresh copy of this piece of data is made with a reference counter set to one.
3. Data is only created and destroyed. It is never changed.
4. To the user, there is no difference between two expressions being equal, but stored in different parts of memory, and two expressions being two representations of the same parts of memory.
5. There is no facility for named fields within a data structure.

Leaving one of these features out would have severe consequences for the other features. (2) explains why (1) is true. (3) ensures that (2) is workable: if two pieces of data are created equal they will remain so. This, in turn, explains why (4) is true. (5) almost follows from (3): in any full fledged programming language, named fields allow all types of actions on the named parts of a data structure that are allowed on whole data structures, that is: creation and destruction. But the possibility to destroy only part of a data structure means that the data structure as a whole is changeable, which violates (3).

In the current version of Bracmat, with objects, the last restriction (5) no longer exists. As a consequence, restrictions (3) and (4) are not true for objects. (2) needs the additional

remark that the reference counter for objects is made so big (counting to more than 1000000000000 before overflowing), that overflow is practically ruled out. Restriction (1) is still true, which means that the Bracmat programmer must take care of the deletion of some pathological structures (to be precise: circular structures, which were non-existent in the object-less Bracmat).

How can a programming language that is created for handling data structures do without named fields? If I only can create and destroy data, how can I let data evolve gradually, piecemeal? The answer lies in the well-developed pattern matching mechanism. Change of data is a two step process. In the first step you retrieve, by means of pattern matching, all those parts of the data that you want to keep. The second step is building the new data from the retrieved parts, together with new pieces. Creation of complex data structures from parts is straightforward in Bracmat. For example, the variable `Row` is a list of three words. We want to change the second word into the word `cat`:

```
{?} the dog runs:?Row      { initial creation }
{?} !Row:%?one % %?three   { step 1: retrieval of 1st and 3rd word }
{?} !one cat !three:?Row   { step 2: reconstruction }
{!} the cat runs
```

This may seem complicated and cumbersome, but look at this:

```
{?} 123:?a                    { create three variables a, b and c }
{?} a sentence:?b
{?} (a.silly,data*structure):?c
{?} (!a.!b.!c):(?b.?c.?a)   { permutation of 3 values in just 1 "statement"}
```

Bracmat has no clear genealogy, but it has borrowed features from a number of programming languages. It is not declarative, like Prolog, nor deeply object oriented, like Smalltalk, but it is more or less procedural, like the majority of languages. Below, I have tried to make the origin of some details more explicit:

C

- conditional execution of right operands of "and" and "or"
- parameter passing (only "by value")

Pascal

- locally defined functions and routines

Lisp

- implementation of expressions in binary trees
- weak type checking
- late binding

Logo

- different notations for the same variable, depending on whether it produces or receives a value

Snobol, Icon

- a well developed pattern matching apparatus including backtracking
- dual results of evaluations: both value and success/failure

Every similarity to other computer algebra systems is a matter of evolutionary convergence.

1.6 Why the name "Bracmat"?

The name Bracmat stems from Ludvig Holbergs novel "Nicolai Klimii iter subterraneum", the history of Niels Klim who visits the planet Nazar and comments the habits of its inhabitants. For example, the people of the country Bracmat are clumsy juniper trees and Niels Klim initially thinks they must be more or less blind. But he discovers that their sight is so sharp that they can see the smallest details in the far distance and therefore don't see what is right in front of them. They are mostly occupied with astronomy and transcendental philosophy, but the state also uses them to do prospection for minerals in the mines.

It so happens that the Latin word "brachiatus," meaning "with arms" or "with branches", in literature of botany often is shortened to "brachiat." I like to think that Ludvig Holberg, who wrote the story in 1741, knew about the works of Carl Linnaeus (1707–1778) and was inspired by him to populate his planet with smart trees. The word "bracmat" is easily morphed into the word "brachiat," and there is at least one case that can be found on the internet where the Optical Character Reader has misinterpreted "brachiat" as "bracmat," caused by a black speck that almost connects the "h" and the "i" in the word "brachiat!" Whether true or not, this derivation highlights one of the most prominent characteristics of the programming language Bracmat: it is all about tree branches.

1.7 How to obtain Bracmat

There are a few download options. You can get the last revision from `https://github.com/BartJongejan/Bracmat`.

At `http://cst.dk/download/bracmat/` you can follow the history of Bracmat from its inception somewhere in the eighties up to the last version. At this site you can also find compiled versions for some platforms.

Alternatively, you can obtain a copy of Bracmat by sending an e-mail to me (Bart Jongejan) at `bart[AT]cst.dk`. Please state your hardware/OS.

1.8 Where to find example code on the internet

At `http://rosettacode.org/wiki/Category:Bracmat` over 100 tasks are solved using Bracmat. The site is an excellent way to compare how the same task is solved in a great number of programming languages.

2 Pattern matching

The single most outstanding feature of Bracmat is how it can recognise patterns in data. The data can be an algebraic expression, a directory listing in table form, a thesaurus structured like a tree, a text or whatever data that can be expressed as a string of characters or as a tree of such strings. Bracmat's patterns are much more advanced than regular expressions. Regular expressions are fixed patterns once the matching operation is started. Bracmat implements not only propositional rules comparable to regular expressions, but also first order predicate rules, backtracking to search the space of combinations of data that make its predicates come true. In this way it is very easy to implement a relational database. But it doesn't stop here, because Bracmat supports recursive invocations of pattern matching and other operations.

2.1 Gentle introduction to pattern matching

The following is an expression that tests whether a specific detail is present in a list of nodes, a sum in this case:

```
{?} 20+a+b+c:?+b+?  { is there a term b in the sum (20+a+b+c)? }
{!} 20+a+b+c
    S   0,00 sec
```

This is what happened. The match operator `:` has a left hand side which is the subject of the pattern, which is right hand side of the match operator. If the match operation is to succeed, the pattern must be similar to the subject. This is the case here, because the question marks are similar to anything (like the Joker in some card plays) and the `b` is similar to itself. By the way, when a pattern match operation succeeds, the resulting expression is the subject. The second example shows a pattern match operation that fails:

```
{?} 20+a+b+c:?+q+?  { is there a term q in the sum (20+a+b+c)? }
    F
```

The `q` in the pattern is not similar to anything in the subject `20+a+b+c` and thus the pattern match operation must fail.

The following examples are quite similar to the introductory example, merely replacing the `+` operator by the `*` and the blank operators:

```
{?} 20*a*b*c:?*b*?  { is there a factor b in the product (20*a*b*c)? }
{!} 20*a*b*c
    S   0,00 sec
{?} 20 a b c:? b ?  { is there a word b in the sentence (20 a b c)? }
{!} 20 a b c
    S   0,00 sec
```

In the following example, the pattern applies to the characters in the string, rather than to nodes in a list of terms, factors or words. The `@` that is prefixed to the match expression indicates to the match operator that this is not the normal match operation, but a string match operation.

```
{?} @(20abc:? b ?)   { is there a character b in the string 20abc? }
{!} 20abc
    S   0,00 sec
```

The patterns in string match operations are similar to patterns in normal match operations that have sentences (words separated by blank operators) as the subject.

2.2 Rewriting data using pattern matching

An important application of pattern matching is rewriting data by means of a number of rules. Each rule consists of a pattern and a replacement expression. Often, the pattern explicitly specifies only parts of the subject, so that the same rule will fire for a broad class of subjects, all having the specified parts of the rule in common. The remaining, unspecified, parts normally have to reappear unscathed in the output of the transformation rule. To catch these parts, patterns can contain any number of variables that bind to the parts of the rule that are not completely fixed.

Suppose we have the sentence `My name is Ivan the terrible` and we want to change `Ivan` to `Wanja`. The following instructions can do that:

```
{?} My name is Ivan the terrible:?begin Ivan ?end
{!} My name is Ivan the terrible
{?} !begin Wanja !end
{!} My name is Wanja the terrible
```

A question mark indicates that the symbol that it precedes is a variable that accepts (part of) the subject as value. An exclamation mark indicates that the following symbol is supposed to be a variable and that the value of that variable must replace the variable.

The pattern `?begin Ivan ?end` and the replacement `!begin Wanja !end` are not sensitive to what the rest of the sentence looks like, so we can use the same pattern and replacement expression to transform another sentence:

```
{?} Ivan goes to school:?begin Ivan ?end
{!} Ivan goes to school
{?} !begin Wanja !end
{!} Wanja goes to school
```

To make a rewrite rule out of these two components, we can put them in a function that we can feed any sentence and that returns a transformed sentence, if the pattern matched the input. In addition, the function below simply returns the input if the pattern didn't match.

```
{?} ( ivanRule
    =   begin end
      .     !arg:?begin Ivan ?end
          & !begin Wanja !end
        | !arg
    )
```

```
{!} ivanRule
{?} ivanRule$(She bought a bicycle for Ivan)
{!} She bought a bicycle for Wanja
```

2.3 Non-linear patterns

Bracmat patterns not only can contain receiving variables (those that are preceded by a question mark), but also giving variables (those that are preceded by an exclamation mark). The same variable symbol can even occur in both roles: as receiving and as giving, in any order. If an occurrence of a variable is receiving and a later in the pattern giving, than the pattern is said to be non-linear. Such patterns can for example be used to find things that occur more than once in the subject. For example, find two inventions of the same age in a list of name/year pairs:

```
{?} (   (teabag.1904) (sonar.1906) (computer.1941)
        (triode.1906) (zeppelin.1900)
      : ? (?invention1.?year) ? (?invention2.!year) ?
    & !invention1 !invention2
    )
{!} sonar triode
```

2.4 Pattern matching with recursive patterns

Many real life problems can be solved by first solving a less complex problem. If needed, this process can be done again and again, until we end with a problem that we know how to solve. In such cases, Bracmat's ability to handle recursive patterns can be of much help.

Here is a rather academic example that defines a grammar with recursive patterns and that checks whether an input is valid, according to this grammar.

```
{?} S=(|0 !S|1 !T);T=(0 !T|1 !S);   { regular grammar }
{?} 0 1 0 1 0:!S   { check whether subject contains an even number of 1's }
{!} 0 1 0 1 0
{?} P=(|0 ?x 1 & !x:!P);   { context free grammar }
{?} 0 0 0 1 1 1:!P   { check whether subject consist of a row of 0's
                       followed by a row of 1's of the same length }
{!} 0 0 0 1 1 1
```

The following example could be the basis for a journey planner. We use a recursive pattern to find out whether two continents are connected over land.

```
{?} connected=("South America"."North America") (Africa.Asia) (Asia.Europe);
{?} (reachable = a b f
    .     !arg:(?a.?b.?f)
        & !f:? ((!a.!b)|(!b.!a)) ?
      |   !f:?A ((!a.?c)|(?c.!a)) ?Z
```

```
        & reachable$(!c.!b.!A !Z)
    );  { remove used fact from fact base }
{?} (   Antarctic Europe Australia Africa Asia "North America" "South America"
    :   ?
        %@?x  { pick a continent }
        ?
        ( %@?y  { pick another continent }
        & reachable$(!x.!y.!connected) { are they reachable? }
        & out$(!x "is reachable from" !y)
        & ~  { force backtracking to collect all answers }
        )
        ?
    ); { pattern using second order logic }
Europe is reachable from Africa
Europe is reachable from Asia
Africa is reachable from Asia
North America is reachable from South America
```

2.5 Pattern matching in strings

`@(`*string* `:` *pattern*`)`

Match *string* with *pattern*.

Pattern matching in a string of characters (a single atom) is like pattern matching in a string of atoms. Use the `@` to instruct the program to look inside the atom and use space operators to combine subpatterns. The space operator does not itself match any characters. To match a space in an atom, use a space in an atom!

You cannot negate the result of string pattern matching by adding the `~` prefix.

```
{?} a b:(~@(? b:a %)) { succeeds, ~@ means: ‘‘not an atom’’}
{!} a b
    S

{?} ~@(a b:a ?)       { succeeds, ~@ means: ‘‘not a string match’’}
{!} a b
    S

{?} @(a b:a ?)        { illegal, LHS of string match operator
                        must be atomic }
{ Bracmat exits }

{?} ~@(a:b)           { fails, pattern matching }
    F

{?} 12/34:@(?x:#?a (~#%@:?y) #?b) { succeeds,}
        { ?x matches the atom 12/34, while #?a (~#%@:?y) #?b
          matches 12/34 as a string }
```

```
{!} 12/34
    S
{?} !a
{!} 12
    S
{?} !b
{!} 34
    S

{?} 12:~/@(?x:#%?a #%?b)  { succeeds, ~ negates /, not @,
                            so we have a string match }
{!} 12
    S
{?} !x
{!} 12
    S
{?} !a
{!} 1
    S
{?} !b
{!} 2
    S
```

2.6 Matching a number in a string

In a string match, the % can be used to force characterwise matching if the subject is a number and the pattern otherwise would have been treated as a number. You have to take care with minuses: the patterns %"-20/5" and %-20/5 are different. In %"-20/5", the % is superfluous and the pattern matches characterwise. In %-20/5, the pattern matches 20/5 and the minus is ignored!

```
{?} @(abcd40/10efgh:?a  20/5  ?z)    { succeeds, because 4 = 20/5 = 4 }
{!} abcd40/10efgh
    S   0,00 sec
{?} !a !z
{!} abcd 0/10efgh
    S   0,00 sec

{?} @(abcd52/13efgh:?a  20/5  ?z)    { succeeds, because 52/13 = 20/5 = 4 }
{!} abcd52/13efgh
    S   0,00 sec
{?} !a !z
```

```
{!} abcd efgh
    S   0,00 sec

{?} @(abcd40/10efgh:?a  %20/5  ?z)    { fails }
    F

{?} @(abcd-20/5efgh:?a %"-20/5" ?z)   { succeeds }
{!} abcd-20/5efgh
    S   0,00 sec
{?} !a !z
{!} abcd efgh
    S   0,00 sec

{?} @(abcd-20/5efgh:?a  %-20/5  ?z)   { succeeds, a = abcd- }
{!} abcd-20/5efgh
    S   0,00 sec
{?} !a !z
{!} abcd- efgh
    S   0,00 sec

{?} @(abcd-20/5efgh:?a  -20/5  ?z)    { succeeds, a = abcd }
{!} abcd-20/5efgh
    S   0,00 sec
{?} !a !z
{!} abcd efgh
    S   0,00 sec
```

2.7 Binary operators in pattern matching

subject : *pattern*

Match *subject* with *pattern*.

A match succeeds if *subject* succeeds and *pattern* is successfully matched with *subject*. The returned value is the left operand, *subject*.

Patterns may be built up from subpatterns and may also include actions that are triggered if a subpattern successfully matches (part of) the subject.

As with the evaluation of other binary operators, the left operand of the `:` operator is evaluated first. The other operand, *pattern*, is not evaluated, but in the process of pattern matching (parts of) *pattern* may be evaluated several times. This is the case with function calls, atoms with a `!` prefix and all right hand sides of the `&` operator. The use of the involved binary operators (`$`, `'` and `&`) and prefixes (`!` and `!!`) as "meta operators" does not restrict the range of matchable expressions in a serious way, as these operators and prefixes normally do not occur in evaluated subject expressions. The same is true

for some other operators (`:`, `|`, `_`, and `=`). These operators, too, have a special meaning within patterns. All other binary operators occurring in a pattern are searched for in the subject expression as part of the pattern matching.

Especially the `&`, `|` and `:` operators are helpful in formulating complex patterns with alternatives, conjunctions and side effects in the form of actions. In the following examples, `!s` stands for the subject expression, the expressions in parentheses are patterns and `!p`, `!pa`, `!pb`, etc. are subpatterns therein. `!a`, `!aa`, etc., stand for an action (a part of the pattern that is conditionally evaluated).

```
!s:(!p&!a)
```

If `!p` matches successfully with `!s`, then `!a` is evaluated. If `!a` fails, the whole match fails. In more complex patterns, only part of the match might fail, resulting in backtracking and retry.

```
!s:(!pa|!pb)
```

If pattern `!pa` does not match with subject `!s`, then `!pb` is tried.

```
!s:(!pa:!pb)
```

If pattern `!pa` matches with `!s`, then pattern `!pb` is also tried.

The next example combines these operators in a grammar-like expression:

```
!s:( !pa          & !A   { if either !pa, or !pb or both of !pc1 and !pc2 }
   | !pb          & !B   { fire, actions !A, !B and !C, respectively,}
   | (!pc1:!pc2) & !C   { are triggered }
   )
```

Notice the grouping of the `:`, `&` and `|` operators:

`(!s:!pa):!pb` and `!s:(!pa:!pb)` have, incidentally, the same effect, but the following expressions are very different:

`(!s:!p)&!e` or `!s:!p&!e` : If `!s` matches with `!p`, `!e` is returned.

`!s:(!p&!a)` : If `!s` matches with `!p`, `!a` is evaluated, but the expression as a whole returns `!s`.

`(!s:!p)|!e` or `!s:!p|!e` : If `!s` matches with `!p`, `!s` is returned. Otherwise, `!e` is returned.

`!s:(!pa|!pb)` : If `!s` matches with either `!pa` or `!pb` (in that order), `!s` is returned.

The possibility that `!s` might fail further complicates the above examples.

2.8 Escaping operator in patterns

Some operators can not be part of a pattern unless "escaped," because these operators play an active role in pattern matching instead of being passive part of a pattern. These operators are `= | & : ' $ _`

The normal role of these operators is ignored by the pattern matching evaluator if they are escaped with a `$` node with an empty LHS.

```
{?} (=foo'bar):(=$(foo'bar))
{!} =foo'bar
{?} (=foo'bar):(=$(?f'?x)) & !f !x
{!} foo bar
```

The escape operator only affects the top node of the escape operator's RHS. The LHS and RHS of the affected node are matched against the subject in the normal way.

```
{?} vowel=.!sjt:(a|e|i|o|u|y)        { function to be used in a pattern to check
                                       that the subject is a single vowel }
{?} (=a$123):(=$((vowel')$(#:?n))) { only the $ between ) and ( is escaped,
                                       because that is the top node of
                                       (vowel')$(#:?n) }
                                     { so the pattern expressions vowel' and #:?n
                                       are evaluated as normal }
{!} =a$123
{?} !n
{!} 123                              { this proves that the RHS of the top $
                                       was evaluated }
{?} (=b$456):(=$((vowel')$(#:?m))) { this fails, proving that the pattern
                                       expression vowel' doesn't recognise
                                       the b in the subject as a vowel;
                                       no assignment to variable ?m takes
                                       place }
{?} (=(vowel')$456) : (=$(($(vowel'))$(#:?m)))     { this succeeds, because
                                       the ' in the pattern expression $(vowel')
                                       is escaped; so the pattern $(vowel')
                                       matches the subject vowel' }
{!} =(vowel')$456
{?} !m
{!} 456
```

The escape operator functions with all Bracmat operators, but the operators `.`, whitespace, `+`, `*`, `^`, `\L` and `\D` should not be escaped normally.

3 The grammar of Bracmat

Note: in long lists the vertical bar `|` is left out.

```
<input>           ::= [<expression>] [; <input>]
<expression>      ::=   <whitespace> <expression> <whitespace>
                      | [<prefixes>] ( <expression> )
                      | <leaf>
                      | <expression> <binop> <expression>
<leaf>            ::= [<prefixes>] <atom-or-nil>
<atom-or-nil>     ::= <atom> | <nil>
<atom>            ::= "<string>" | <string>
<string>          ::= <character> [<string>]
<character>       ::= any printable character except \ and " | <spec>
<spec>            ::= \a \b \t \n \v \f \r \" \\
```

```
<nil>            ::= ""   (or nothing at all, such as in "()")
<binop>          ::= = . , | & : <whitespace> + * ^ \L \D ' $ _
<prefixes>       ::= <prefix> [<prefixes>]
<prefix>         ::= [ ~ / # < > % @ ` ? ! !!
<whitespace>     ::= spaces, tabs, new line and form feed characters
```

Whitespace (operator/cosmetic measure) almost never leads to confusion. It does in (some) cases where a *nil* leaf without prefixes is adjacent to the whitespace operator. For example: `get' out$now`. Bracmat interprets this as `get'(out$now)`. `""` or `()` fixes the problem: `get'() out$now`. Quotation marks are not part of the string they surround. They should be used if necessary, e.g. `in this case` or `he{this is not a comment}re`. Comments can be written everywhere, except in the middle of a string in quotation marks. Comments are enclosed in `{}` and may be nested.

4 Binary operators

4.1 Overview

`= . , | & :` whitespace `+ * ^ \L \D ' $ _`

These are the 15 binary Bracmat operators. The higher in the list, the lower in the order of operations. Use parentheses to overrule the ordering of precedence and force an operator to a higher position in the order of operations, as in:

```
{?} (a+b)*(a+c)+a^(-1*d^2+(d+1)*(d+-1))
{!} a^-1+a^2+a*b+a*c+b*c
```

`=`

assignment: `(x=7) (square=.!arg^2)`

object member definition:
`(myobject=(place=Copenhagen) (setPlace=.!arg:?(its.place)))`

`.`

fixed data structure: `(Palme.Olof.Sweden)`

object member referencing:
`(myhash..insert)$(xxx.998743)`

`,`

list with autostretch:
`(ham,(bread,butter),jam):(ham,bread,butter,jam)`

`|`

or else: `get$myfile | out$"Cannot read myfile"`

`&`

and then: `out$"almost done" & Done`

`:`

match subject with pattern:
`Meeting at 4 in room 24:? ?#hour ? ?#room ?`

whitespace

sentence, neutral element:
`Oh well this can go on and on:?a Oh ?z & !a:`

`+`

add, neutral term 0: `(a+6+b+a+10)`, `(a+b:?x+a+?y & !x:0)`

`*`

multiply, neutral factor 1: `(46546*647547564)`, `(a*b:?x*a*?y & !x:1)`

`^`

raise to a power, neutral exponent 1: `(9975^332)`, `(45:?n^?exp & !exp:1)`

`\L`

take logarithm: `10\L1050`

`\D`

differentiate: `x\D(x^10)`

`'`

function evaluation (does not evaluate RHS): `str'(b+a):"b+a"`

macro: `()'(my name is ()$name)`

`$`

function evaluation (evaluates RHS): `str$(b+a):"a+b"`

variable in macro: `()'(my name is ()$name)`

`_`

dummy: `!expr:?lhs_?rhs & (do$!lhs)_(do$!rhs)`

4.1.1 The = operator

This operator ensures that the right hand operator stays unevaluated. It is mainly used in the definition of pieces of code (e.g. functions). The code on the right is bound to the name on the left.

atom `=` *expression*

Each time when the value of *atom* is asked for, a fresh copy of *expression* is made available. *expression* itself is unchangeable and can only be wiped out by removing the binding between *expression* and its name, *atom*. This has, in turn, no influence on the copies made earlier.

```
{?} a=2      { create binding }
{!} a
    S
{?} !a:?b    { bind copy to b }
{!} 2
    S
{?} !b       { show b's value }
{!} 2
    S
{?} a=3      { remove a's binding to 2 }
{!} a
    S
{?} !b       { show b's value }
{!} 2
    S
```

There is a second way of using the = operator, with a slightly different syntax:

nil = *expression*

The = operator serves as a shock proof container for *expression*. The effect of evaluating this type of expression is almost the same as that of the macro instruction `()'`*expression*. Indeed, after evaluating a macro instruction we have an expression with the *nil* = *expression* syntax.

```
{?} out$(b+a)
a+b
{?} out$(=b+a)
=b+a
{?} out$('(b+a))
=b+a
{?} c=3
{?} out$(=b+a+$c)
=b+a+$c
{?} out$('(b+a+$c))
=b+a+3
```

4.1.2 Differentiation

variable `\D` *expression*

Bracmat knows how to differentiate expressions in which no other binary operators occur but `+ * ^` and `\L`.

Example:

```
{?} y\Dx\D(a^(x^2+y^2))
```

```
{!} 4*a^(x^2+y^2)*x*y*e\La^2
{?} y\Dr
{!} 0
```

The last example gives zero, which in many applications isn't what we want. Often, with `y` we express the y-component of a vector with length r, and r consequently is a function of y (and the other components). We can solve this as follows:

```
{?} dep=(r.x) (r.y) (r.z) { dep is a special variable }
{?} y\Dr
{!} y\Dr
```

Now the expression is just left unevaluated. Later, you can substitute an expression for `r` in terms of its components

```
{?} y\D(r^-1):?derivative
{!} -1*r^-2*y\Dr
{?} sub$(!derivative.r.(x^2+y^2+z^2)^1/2):?derivative
{!} -1*y*(x^2+y^2+z^2)^-3/2
```

And, if you like, you can simplify the result by putting `r` back in:

```
{?} sub$(!derivative.x^2+y^2+z^2.r^2):?derivative
{!} -1*r^-3*y
```

4.2 Assignment to variables

There are two forms of assignment to a variable:

variable `=` *expression*

: *expression* is not evaluated before assignment to *variable*.

expression `:` `?`*variable*

: *expression* is evaluated before assignment takes place.

The `=` operator is used to bind (still) unevaluated expressions such as patterns and functions to variables.

Assignment with the `:` makes use of pattern matching with a universally unifying pattern. This way of assignment is very powerful and can even be used to assign unevaluated expressions, by preceding the subject with an `=` or an `'` operator.

Example: define Lisp's car-function, first using `=` and then using `:` to bind the function definition to the variable `car`.

```
{?} car=.!arg:(?%arg ?)&!arg            { one may freely reuse arg! }
{?} (=(.!arg:(?%arg ?)&!arg)):(=?car) { another way to define car }
{?} car$(one two three)
{!} one
{?} (four five six):(?`%first ?rem)    { `: 0 of 1, %: 1 or more, together 1 }
```

```
{?} The first element is !first and the remainder is !rem
{!} The first element is four and the remainder is five six
```

4.3 Binary operators in program flow

exprA `&` *exprB*

(*exprA* and then *exprB*) *exprB* is only evaluated if *exprA* succeeds,

exprA `|` *exprB*

(*exprA* or else *exprB*) *exprB* is only evaluated if *exprA* does not succeed.

In both cases *exprA* is always evaluated and *exprB* conditionally. If *exprB* is to be evaluated, *exprA* and the `&` or `|` operator have served their purpose. Therefore, they are eliminated before *exprB* is evaluated. In this way, the program stack doesn't grow indefinitely when recursive calls are made from the right hand side of any `&` or `|` operator occurring in an expression. Even a conventional sequence of instructions (where the success or failure of the evaluations of each instruction do not matter) can make use of this tail recursion optimisation. In that case one uses the pacifier (short cut prefix) `‘`.

```
(‘!a & !b)      !b is always evaluated. (sequence)
(‘!a | !b)      !b is not evaluated. (useless in this form)
```

The pacifier or shortcut prefix is inherited by higher levels, it percolates towards operators that are closer to the root of the tree, until it is subsumed in situations like the above ones.

4.4 Algebraic operations

term `+` *term*

addition

factor `*` *factor*

multiplication

base `^` *exponent*

exponentiation

base `\L` *expr*

logarithm

variable `\D` *expr*

differentiation

Subtraction and division are treated as special forms of addition and multiplication. Therefore there are no binary operators for subtraction and division. (The minus sign `-` and the slash `/` can be used in numbers, however.)

If one operand of an algebraic operator is evaluated then the other one is normally evaluated as well, even if this may seem unnecessary (multiplication by 0). This is done to ensure that all side effects take place as intended. However, if an operand fails to evaluate then the algebraic expression fails too and if the failing operand is the left hand side of the expression, then the right hand side is not evaluated. In this sense algebraic operators behave like the logical `&` operator.

Bracmat gives the user practically *no* control over the format of evaluated algebraic expressions, such as the order of terms or factors. Bracmat tries to present algebraic objects in a unique (canonical) form. This is in many cases an unattainable goal: the forms

```
(a+b)*(c+d)
```

and

```
a*c+a*d+b*c+b*d
```

are both stable expressions. On the other hand,

```
(a+b)*(c+d)+e
```

becomes

```
e+a*c+a*d+b*c+b*d
```

Bracmat keeps completely factorised expressions as they are, because factorization is an expensive operation. For the same reason, Bracmat does not automatically factorise factorisable expressions. Another domain of duality are expressions with logarithms.

Sums and products start with rational numbers, followed by `pi`, `i` and `e` (if present, that is). Then follow other terms and factors. It is recommended not to assume anything about the ordering of these terms and factors, as this may change in later versions of the program.

4.5 Function evaluation

The binary operators `$` and `'` are similar in most respects. In general, the left operand evaluates to the name of a built-in or defined function, whereas the right operand is an expression that is passed as an argument to the function. The `$` evaluates the right operand before it is passed over, the `'` doesn't. Parameter passing is by value, although the implementation postpones and limits copying of data as much as possible. In the code of the called function, the passed argument is bound to a local variable that is always called `arg`.

Most often, the left operand of the `$` and the `'` operator evaluates to an alphanumeric name. There are a few special function names:

- No name at all. Here, the `$` and the `'` operator have decidedly different and complementary roles. (Forced evaluation of subexpressions in otherwise unevaluated expressions, such as patterns.)
- An integral number. (Array indexing.)

- Only prefixes. (Prefix pasting or influencing success and failure of non-atomic expressions.)

Function calls are even effective in patterns, as it is fair to assume that the `$` and `'` operators seldom occur in subjects and so need not to be matched (the same is, a fortiori, true for the `&` and `|` operators). In patterns, the return value of a function is part of the pattern. A function may be called several times during one evaluation of a matching expression, due to backtracking and retrying.

4.6 Macro evaluation

The `'` operator with empty LHS is the macro evaluator. The macro evaluator returns the RHS unchanged, except where `$` operators with empty LHS occur.

The expression to the RHS of such a `$` operator must evaluate to the name of a variable or to a definition. The whole `$` expression is then replaced by the value of the variable or the definition. (The following example illustrates three ways how you can make sure that the LHS of the `$` operator is empty.)

```
{?} x=un
{?} y=(member=deux)
{?} '(one ($x), two ""$(y.member), three ()$(=trois))
{!} =one un,two deux,three trois
```

Here is an example that shows that the RHS of the `$` operator can be an expression that first needs to be evaluated:

```
{?} a=eins
{?} b=zwei
{?} dice=clk$
{?} '(Eins oder zwei? ()$(!dice:<1&a|b))
{!} =Eins oder zwei? eins
    {...}
{?} '(Eins oder zwei? ()$(!dice:<1&a|b))
{!} =Eins oder zwei? zwei
```

You can construct any Bracmat expression using macro evaluation. For example, the function `F`, below, transforms an expression with `+` and `*` operators to a pattern, replacing any non-empty atom that is not a number to a pattern variable that either is receiving or giving, depending on whether the atom occurs for the first time or not, when reading the expression from the beginning to the end.

```
{?} (F=
      seen f
    .   :?seen
      & ( f
        =   a b
          .   !arg:%?a+%?b&'($(f$!a)+$(f$!b))
```

```
                | !arg:%?a*%?b&'($(f$!a)*$(f$!b))
                |     !arg
                    : (
                      | #
                      | e
                      | i
                      | pi
                      )
                  & '$arg
                | !seen:? !arg ?&glf$('(!.$arg))
                | !arg !seen:?seen&glf$('(%?.$arg))
            )
          & f$!arg
        );
{?} F$(a*b+3*a*c)
{!} =%?a*%?b+3*!a*%?c
```

Now we can use the pattern to match expressions with the same structure and with the same operators

```
{?} x*y+3*x*z:!(F$(a*b+3*a*c))
```

The function F can only handle expressions containing `+` and `*` operators. Now we will in a few steps amend F so that it can handle any operator. This is done by using the dummy `_` operator.

As a first step, we will have to protect the argument of F against evaluation, because otherwise we will never be able to analyse expressions that evolve to different expressions when evaluated. So, from now on we will use F with the `'` operator, like so.

```
{?} F'(a*b+3*a*c)
```

To simplify the first amendment, we will only handle expressions with the `+` operator. Later we will generalize to handling any operator.

```
{?} (F=
      seen f
    .   :?seen
      & ( f
        =   a b g
          .     !arg:(=%?a+%?b)
              & '($(f$('$a))+$(f$('$b)))
            |   !arg:(=?g)
              & (     !g
                    : (
                      | #
                      | e
```

```
                        | i
                        | pi
                        )
                    & !arg
                  | !seen:? !g ?&glf$('(!.$g))
                  | !g !seen:?seen&glf$('(%?.$g))
                  )
          )
        & f$('$arg)
    );
{?} F'(a+b+pi+a)
{!} =%?a+%?b+pi+!a
```

Now we replace the `+` operator by the dummy operator `_`. If we only do that, then we will only see `_` operators in the result. That is because in a macro context, a `_` operator is not evaluated to its current value. However, if the RHS expression of a `$` operator with empty LHS is headed by the `_` operator, then the `_` operator is evaluated to its current value, analogous to what happens if the RHS were the name of a variable. So we add an extra `$` operator.

When the `_` operator is evaluated to its current value, the LHS and the RHS are also evaluated.

```
{?} a^b:?_?
{?} LHS:?left
{?} '$(($left)_($(=RHS)))
```

Without evaluation of `_` operator:

```
{?} '($left)_($(=RHS))
{!} =LHS_RHS
```

With evaluation of `_` operator:

```
{?} '$($left)_($(=RHS))
{!} =LHS^RHS
```

Now we use this in the function `F`:

```
{?} (F=
      seen f
    .   :?seen
      & ( f
        =   a b g
          .     !arg:(=%?a_%?b)
              & '$($(f$('$a)))_($(f$('$b)))
          |   !arg:(=?g)
            & (     !g
```

```
                    : (
                      | #
                      | e
                      | i
                      | pi
                      )
                  & !arg
                | !seen:? !g ?&glf$('(!.$g))
                | !g !seen:?seen&glf$('(%?.$g))
                )
          )
        & f$('$arg)
    );
{?} F'(a+b_k&out$y&$$(=.arg))
```

Finally, we ensure that atoms that already have a prefix are treated in the same way as empty strings and numbers.

```
{?} (F=
      seen f
    .   :?seen
      & ( f
        =   a b g
          .     !arg:(=%?a_%?b)
              & '$($(f$('$a)))_($(f$('$b)))
            |   !arg:(=?g)
              & (   (   !g
                      : (
                        | #
                        | e
                        | i
                        | pi
                        )
                    | ~(glf$('(~`/#<>%@?!.$g)))
                    )
                  & !arg
                | !seen:? !g ?&glf$('(!.$g))
                | !g !seen:?seen&glf$('(%?.$g))
                )
          )
        & f$('$arg)
    );
```

```
{?} F'('(the hare&$the tortoise))
{!} ='(%?the %?hare&$!the %?tortoise)
```

In the above example the argument of the `F` function is returned in an unevaluated state, though transformed to something else. The last input was a macro expression, and the result is still a macro expression.

Now suppose that we want to produce the macro expression

```
'(There are 23 elements in list ()$L)
```

by means of another macro expression that evaluated the number of elements to 23. Something like this:

```
{?} 23:?N
{?} '('(There are ()$N elements in list ()$L))
```

This generates the error "macro evaluation fails because rhs of `$` operator is not bound to a value: `$L`". What is needed is to 'escape' the `$` operator in the subexpression `()$L`, since we do not want `()$L` to be evaluated. This is done with and extra `$` operator, similar to how a backslash is escaped in many programming languages:

```
{?} 23:?N
{?} '('(There are ()$N elements in list ()$($L)))
{!} ='(There are 23 elements in list ()$L)
```

More examples:

```
{?} '(b+a c)
{!} =b+a c
{?} (x=value) & '(a ($x) z)
{!} =a value z
{?} (object=(member=value)) & '(a ($(object.member)) z)
{!} =a value z
{?} '(a ($(=value)) z)
{!} =a value z
```

4.7 Program transformation

Bracmat code is data. Yet it is easier to transform the internal representation of JSON or XML data than to transform a Bracmat program. The reason is that for code introspection it is necessary to turn off the expression evaluation that normally takes place in patterns. This can be achieved by using macros. Here is an example that reverses (`rev$`) and lowercases (`low$`) all leaves, but keeps everything else the same: prefixes, operators—even those that play special roles inside patterns, such as the `!` prefix and the `_` operator.

```
{?} ( tr
    =   a b f
      .   flg$!arg:(=?f.?arg)
        &   glf
```

```
            $ (
              ' ( $f
                .
                  $ (   '$arg:(=?a_?b)
                      & '$($(tr$('$a)))_($(tr$('$b)))
                    | low$rev$!arg:?arg&arg
                    )
                )
              )
    )
{?} tr$(=foo"STRing" ~!(sin$4/56) !and_?<UNDERSCORE)
{!} (=oofgnirts ~!(nis$65/4) !dna_<?erocsrednu)
```

4.8 The dummy operator _

Bracmat has only one variable that binds to a binary operator, the _ operator. Worse even, this variable is global. Nevertheless this variable is most useful in definitions of certain types of recursive functions (tree walkers).

The assignment of a new value to the _ variable can only take place in a match operation. A _ in a pattern is always receiving, whereas a _ outside a pattern is either giving or left unchanged. Try this:

```
{?} a_b              { this has unpredictable results }
{?} x^y:?_? & a_b    { _ gets bound to ^, thus a_b evaluates to a^b }
```

A _ is evaluated by the expression evaluator, but also by the macro evaluator. The latter is useful if the _ has matched an operator that is very volatile, such as & and |.

```
{?} (=!a:!b&!c):(=?left_?right) { match the & }
{?} '$_
{!} =&                          { it worked, the _ is replaced by a & }
{?} get$(str$('$_),MEM,VAP):"=" ?op & !op { freeze and slice }
{!} &            { the operator is immobilised in a string }
```

The _ variable is always expanded BEFORE the left and right hand side operands are evaluated. That explains why new assignments in the operands do not result in unwanted side effects in the upper node with the _.

4.9 Recursion and the _ operator

In Bracmat functions are allowed to call themselves. Often this happens if a function's argument is split into a left subtree and a right subtree and the function is called with each subtree in turn as its argument. If the operator between the subtrees is unknown, it is time consuming to try all patterns ?+?, ?*?, ?$?, ?'? etc. The _ operator circumvents this problem. It is a dummy operator that matches any other operator and expands to

the operator with which it matched last time. Thereby preceding matches are forgotten: the _ operator is a global variable.

```
{?} ( reverse
    =   l,r
      .     (!arg:?l_?r)                { if arg is a compound expression ... }
          & (reverse$!r)_(reverse$!l)   { ... swap the reversed operands }
        | !arg                          { let atoms as they are }
    )
{?} reverse$(Bill loves sweet Nancy. This is true)
{!} true is This.Nancy sweet loves Bill
```

4.10 Some often used control structures

Here are the nearest equivalents of some traditional control structures.

Sequence

```
a; b;
    `!a&!b
    !a !b
    !a,!b
    !a.!b
```

Repetition

```
WHILE a DO b;
    whl'(!a&`!b)
DO b WHILE a;
    whl'(`!b&!a)
FOR i := m TO n DO b;
    !m+-1:?i&whl'(1+!i:~>!n:?i&`!b);
```

Selection

```
IF a THEN b ELSE c;
    !a&`!b|!c;
v := IF a THEN b ELSE c;
    (!a&`!b|`!c):?v { works even if !c fails }
```

Branching

```
CALL a;
    !a;
CALL b(x,y,z);
    b$(x,y,z);
v := b(x,y,z);
    b$(x,y,z):?v;
```

4.11 The nameless functions $*expression* and '*expression*

Sometimes a variable predictably will evaluate to the same value repeatedly, for example in an inner loop or a pattern that repeatedly backtracks. In such situations macro substitution can improve performance by replacing the variable by its value in an early stage.

In Bracmat, a macro has the general form '*expression*. When '*expression* is evaluated, *expression* is searched for subexpressions headed by the operator `$`, with empty LHS. Such subexpressions are replaced, depending on what is found on the RHS of the `$` operator.

After macro substitution has taken place, what remains is an expression of the form `=`*expression*. The `=` operator is a safeguard against evaluation of expression.

Macro substitution makes it possible to dynamically create unevaluated code and bind it to a variable.

```
{?} '($out):?my-fun-var
{?} !my-fun-var$(Hello world)
{?} '($out):(=?my-fun-alias)
{?} my-fun-alias$(Hello world)
```

Pattern matching can sometimes be made more efficient by using macro substitution, but the resulting code is harder to understand:

```
{?} ( 0:?count
    &   41 3 5 7 6 23 12 11 19
     :   ?
         %?`A
         ?
         ( %?`B                          { each number pair A,B ... }
         & !A+!B:?C                      { is added only once, giving C }
         &   '(? ()$(!count+1:?count&C) ?)
           : (=?rem)                     { C's value is hard-coded into rem }
         )
         !rem                            { which is the remaining pattern }
    & out$(after !count "trials:" !A "+" !B "=" !C )
```

```
    )
```

after 16 trials: $5 + 7 = 12$.

In the same way, function code can be pieced together before it is ever executed.

```
{?} power=three
{?} ((!power : two & (=!arg^2)) | (=!arg^3)) : (=?abc)   { if power = two,
      abc is bound to !arg^2 (unevaluated); otherwise, abc is bound to !arg^3 }
{?} '(.!arg + -1*$abc + 2) : (=?poly) { poly is the name of a new function
      that will return a value that depends on the current value of arg
      and on the value of power at the time when abc got its binding }
{?} lst$poly   { show poly's definition }
{?} poly$4
{!} -58
```

The macro construct *'expr* is useful if an expression has to be executed many times while parts of it remain constant, for example in nested loops:

Without macro construct (5×5 multiplication table):

```
{?} 0:?m                          { initialise counter of outer loop }
{?} (outer = 1+!m : <6 : ?m       { code for outer loop }
           & put$\n               { start output on new line }
           & 0 : ?n               { initialise counter of inner loop }
           & `!inner              { execute inner loop }
           & !outer)
{?} (inner = 1+!n : <6 : ?n       { code for inner loop }
           & put$(!m X !n "=" !m*!n ", ")
                                  { the same !m is expanded 10 times }
           & !inner)              { loop }
{?} !outer
```

With macro construct:

```
{?} 0:?m
{?} (outer = 1+!m : <6 : ?m
           & '( 1+!n : <6 : ?n
              & put$($m X !n "=" ()$m*!n ",")
                                  { !m is expanded only 2 times }
              & !inner
              ) : (=?inner)       { at each pass through the outer loop
                                    the inner loop inner is defined anew }
           & put$\n
           & 0 : ?n
           & `!inner
           & !outer)
```

```
{?} !outer
```

5 Objects

With the = and . operators you can construct and dereference conventional data structures and even objects with methods. In an expression, each subexpression with a = operator in the top node and an atom in the LHS of the top node indicates a field or object method that can be accessed and changed independently of other fields and methods, i.e. without the need to dissect and reassemble the whole expression. Such expressions are objects. An object member (a field or method) is addressed by using the LHS of the = operator as the member's name, preceded by the object's name. The name of the object and the name of the member must be separated by a dot operator.

In the example below an object named `John` is created with the members `length`, `age` and `name`. The `name` member has two submembers `first` and `family`:

```
{?} John = (length=180),(age=30),(name=(first=John) (family=Bull))
```

There is no prescribed way in which the members should be glued together to form an object. Here, the comma operator and blank operator are used, but any operator except the = operator can be used to separate field names. John's length can be changed to 185 in the following ways:

```
{?} John.length = 185
```

or

```
{?} 185 : ?(John.length)
```

The same object can be assigned to another variable, creating an alias, but we have to take care not to evaluate `John`, because that would create or overwrite the variables `length`, `age` and `name`):

```
{?} !John:?alias
    { wrong, alias=length,age,name }

{?} '$John : (=?alias)
   { right, alias=(length=185),(age=30),(name=(first=John)(family=Bull)) }
```

Bracmat replaces the expression `'$John` by the value of John, protected against evaluation by a = operator. For that reason, the pattern on the RHS of the match operator : contains a = operator as well. Now we can change John's age by operating on the variable `alias`:

```
{?} alias.age = 31
```

To see that the above expression indeed has the wanted (side-)effect, we can inspect `John`:

```
{?} lst$John
(John=
  (length=180)
, (age=30)
, (name=(first=John) (family=Bull))
```

```
);
```

Alternatively, we can also just show the field `age` in John:

```
{?} !(John.age)
{!} 31
```

It is also possible to create an alias for a subobject. Taking the previous example, we could create an alias for the `name` member:

```
{?} '$(John.name):(=?nm)
```

Now assign a new family name:

```
{?} Flinter:?(nm.family)
{?} lst$John
(John=
  (length=180)
, (age=30)
, (name=(first=John) (family=Flinter)));
```

Using an alias for a subobject can save some code and processing time if the subobject is accessed many times. Without the alias for John's name, we can change his family name in this way:

```
{?} Flinter:?(John.name.family)
```

It is valid to have an empty name for a member:

```
{?} x=(header=blabla) (=(a=1) (b=2))
```

Here, `a` and `b` are fields in a nameless subobject of `x`. We can ask for the value of `b`:

```
{?} !(x..b)
{!} 2
```

To retrieve the whole subobject:

```
{?} '$(x.):(=?subobject)
{?} lst$subobject
(subobject=
(a=1) (b=2));
```

An alias can also be created for part of an object:

```
{?} x=(a=) (b=) (c=) (d=)
{?} '$x:(=(a=) ?alias (d=))
```

Now `alias` only shares the members `x.b` and `x.c` with `x`. The same result follows from

```
{?} '$x:(=? ((b=) (c=):?alias) ?)
```

Objects can be composed to form new objects containing the union of the members of the contributing objects:

```
{?} x=(a=) (b=)
```

```
{?} '((p=) ($x) (q=)):(=?r)
```

Evaluation of an expression that contains = operators can have unexpected side effects, as the following example shows.

First suppose that x (containing one record with one anonymous field) is unevaluated (case A) and assigned to two other variables:

```
{?} x=(=)
{?} !x:?y
{?} !x:?z
```

In this case, x, y and z are different objects. For example

```
{?} 2:?(y.)
```

does not affect x and z. Do the assignment again, but this time evaluating x only once:

```
{?} !x:?y:?z
```

Now y and z are the same object, but still different from x. A change made to y affects z but does not affect x.

Suppose that x *is* evaluated (case B):

```
{?} (=):?x
{?} !x:?y
{?} !x:?z
```

Now x, y and z are the same object.

Explanation: in (A) the value of x is not evaluated, especially the LHS of the = operator. Therefore, a new = node is created each time x is evaluated. In (B), the value of x *is* evaluated, so no new copies of the = node are made.

6 Construction of data structures

In Bracmat, linear lists can be made by separating the elements with comma, plus sign, asterisk, space and dot. The first four operators create linear structures (right descending lists), moving nodes as necessary, whereas the dot operator creates any tree structure. In addition, the plus sign and the asterisk (times) do not preserve the order of the elements if they are not canonical order. Which operator one should use in a given situation depends on the following considerations:

- Space, comma, plus and asterisk offer automatic concatenation of lists, but are slower than the dot.
- Space, plus and asterisk are useful if there is a need to search in a list, because these operators support backtracking, but they are slower than the comma.
- The comma can be used in linear lists that are accessed recursively and in record-like structures, where the elements must have absolute positions. The dot can be used for the same purpose.

- If a function has a fixed number of parameters, you should use dots to separate them.
- If there is a need to sort elements alphabetically, use the plus operator.
- Lists constructed with plus or asterisk behave like sets. Adding an element that already is present does not make the list any longer, but will instead increase a factor (plus) or exponent (asterisk).

Examples:

```
{?} x=a.b.c
{?} y=p.q
{?} !x.!y
{!} (a.b.c).p.q
{?} x=a b c
{?} y=p q
{?} !x !y
{!} a b c p q
{?} set=jan+piet+klaas
{?} !set
{!} jan+klaas+piet
{?} !set+klaas
{!} jan+2*klaas+piet
{?} rotate=car,cdr.!arg:(?car,?cdr) & (!cdr,!car)
{?} rotate$(one,two,three,four)
{!} two,three,four,one
{?} rotate$((one,two),(three,four))
{!} two,three,four,one
```

By combining dots, commas and spaces, one may build any tree-like data structure that, thanks to the backtracking mechanism on space-separated lists, make the formulation of queries (goals) almost as easy as in Prolog. This is an example of a simple database, in which each row starts with a descriptor field, followed by a varying number of similar fields.

```
{?} M=( (odd  ,1 3 5 7 9)
        (even ,0 2 4 6 8)
        (prime,2 3 5 7)
      )
```

We choose the space operator to form the backbone of the lists of numbers, because we want to access these numbers associatively, by using the back-tracking mechanism.

Let us formulate a query that searches for all numbers that occur in two or more categories (odd, even, prime). The findings are to be printed to the screen.

```
{?} ( !M
```

```
:   ?                                  { skip 0 or more rows... }
    (?c1,?row)                         { ... fetch (number type, number row)... }
    ?                                  { ... skip 0 or more rows... }
    ( ?c2                              { ... fetch another number type,... }
    ,   ?                              { ... skip 0 or more numbers... }
        ( %?'el                        { ... fetch a number... }
        & !row:? !el ?                 { does number occur in earlier row? }
        & out$(!el is both !c1 and !c2) { yes? show result }
        & ~                            { not satisfied yet: fail and backtrack }
        )
        ?                              { skip rest of numbers }
    )
    ?                                  { skip rest of rows }
)
```

This prints

```
3 is both odd and prime
5 is both odd and prime
7 is both odd and prime
2 is both even and prime
```

and finally fails when backtracking (induced by the `~`) has found all answers to the query.

Experimentation with the implementation of matrices in Bracmat has revealed that lists (of lists (of lists...)) lead to smaller and faster programs than arrays, artificially made multidimensional by playing with the index. A drawback of the list approach is its unconventionality. Much time has to be spend in reformulating existing algorithms based on indices. On the other hand, the list approach is essentially insensitive to the dimensionality of the matrix at hand, and may even be indifferent to the number of indices.

6.1 Program flow

Most binary operators are used in expressions that flow on their own or flow not at all. In the first group are the arithmetic operators, in the second is the dot operator. In between are the two other structuring operators, comma and whitespace.

Branching to a function is done with the `$` and `'` operators:

`a$b` (or `a'b`) evaluates function `a` with argument `b`.

Branching without argument passing and local variables is done with the unary operator (prefix) `!` but often this prefix and its cousin `!!` are used for the purpose of variable expansion, it just depends on whether a variable is bound to an unevaluated or to an evaluated expression:

`!X`

do subroutine `X`

`!X`

expand `X`

`!!Y`

expand expansion of `Y` (two `!`s is the maximum)

Conditional evaluation is decided by the success or failure of subexpressions. Every (sub)expression has two kinds of value: a visible value and a success(S)/failure(F)/ignore(I) value. Success and failure are primarily decided by the low level functions in the interpreter. The ignore value is generated if a failing expression is back-quoted. The `&` and `|` operators are sensitive to the S/F/I value of the left operand (where I counts as S). Often this left operand is a matching expression.

`!a & !b`

if `!a` succeeds do `!b`

`!a | !b`

if `!a` fails do `!b`

`!subject : !pattern`

try to prove that `!pattern` describes `!subject`

The back quote ‘ can be used to overrule the failure of a subexpression. The tilde ~ negates failure and success.

`‘!p & !q`

do `!p` and then do `!q`

`!a:!p & ‘!b | !c`

if `!a` matches `!p` do `!b` else do `!c`

`~!a`

succeeds if `!a` fails and fails if `!a` succeeds

7 Prefixes

7.1 Prefixes and program flow

Unlike other programming languages, Bracmat does not return the value of a variable or object member if we type its name. In Bracmat, variables and object members have to be told explicitly that we want their value, not their name. This is achieved with the `!` and the `!!` prefixes in front of the variable name or object member name.

`!`*atom*

is replaced by the binding of *atom*

`!!`*atom*

is replaced by the binding of the binding (after evaluation) of *atom*

Likewise `!(`*object-name*`.`*member-name*`)` is replaced by the binding of *object-name*`.`*member-name*.

Bindings can be evaluated or unevaluated. In the last case, the next step after expansion is the evaluation of the binding, unless expansion took place within a pattern.

```
{?} 2+3:?four       { bind 5 to four }
{?} !four           { evaluation has already taken place when four is expanded }
{!} 5
{?} 5=2+!four       { numbers are legal names; 5 is bound to 2 + !four }
{?} !5              { evaluation takes place immediately after expansion }
{!} 7
{?} sum=%+%         { define pattern sum }
{?} a+b+c:!sum      { is a+b+c a sum? after expansion, %+% is not evaluated }
{!} a+b+c
    S   0,00 sec
```

The `!!` prefix is not used as often as the single `!`, but comes in handy if you want to pass a variable by name instead of by value.

```
{?} (check=one,two,criterion
      .     !arg:(?one,?criterion,?two)
          & !!criterion
          & TRUE
        | FALSE
    )
{?} is-greater-than = !one:>!two
{?} is-divisor-of = (div$(!two,!one)*!one):!two
{?} check$(3,is-greater-than,15)   { pass by name }
{!} FALSE
    S   0,00 sec
{?} check$(3,is-divisor-of,15)
{!} TRUE
    S   0,00 sec
```

Passing by name is used here to postpone the evaluation of the second argument until it has arrived in the function `check` and the local variables `one` and `two` have been bound to the first and the third arguments, respectively.

Postponement of evaluation can also be achieved with the `=` and the `'` operators.

```
{?} (chack=one,two,criterion
      .     !arg:(?one,(=?criterion),?two)
          & !criterion
```

```
          & TRUE
        | FALSE
    )
{?} is-greater-than = =!one:>!two                    { an extra = }
{?} is-divisor-of = '((div$(!two,!one)*!one):!two)   { an extra ' }
{?} chack$(3,!is-greater-than,15)                    { pass by value }
{?} chack$(3,!is-divisor-of,15)
```

7.2 Prefixes and pattern matching

In patterns, atoms and expressions within parentheses may be preceded by prefixes that control the matching process.

Below, use the term "trivial elements" for elements that are either neutral or zero. A list expression (an expression that links elements with the space operator) has the empty string as neutral element. In sums, `0` (zero) is the neutral element. In products, `1` is the neutral element. So the empty string, `0` and `1` are trivial elements in lists, sums and products, respectively. In addition, `0` is also a trivial element in products. Lists and sums do not have "zeros."

`!` and `!!`

in front of an non-nil atom or an expression denoting a member of an object causes expansion of the atom or the member to its direct or indirect binding. This binding is matched with the subject.

`‘`

causes backtracking if the pattern did not successfully unify with a non-trivial element of the subject-list. A list is an expression consisting of terms (`+` operator), factors (`*` operator) or words (whitespace operator). In non-sophisticated patterns, `‘` means simply: unify with at most one non-trivial element. Zero non-trivial elements are allowed, in which case unification takes place with an implicit trivial element: Bracmat sees 0's everywhere in a sum, 1's in a product and zero length words in a sentence.

`?`

unifies with anything. If `?` is followed by a non-nil atom denoting a variable or an expression denoting a member of an object, then the matched part of the subject is captured by this variable or member. In other words, pattern matching can have assignment as a side-effect.

`@`

unifies only with atoms. Also, if prefixed to the `:` operator, it indicates that the pattern applies to the characters inside the atom (string match).

`%`

causes the match to succeed only with one or more non-trivial elements of the subject-list. (Exception: in combination with `[` prefix.)

`<`

unifies only with atoms that are less than the atom following the `<` prefix.

`>`

unifies only with atoms that are greater than the atom following the `>` prefix.

`#`

unifies only with rational numbers.

`/`

unifies only with non-integer rational numbers.

`~`

constrains the match to subjects that are not equal to the atom following the `~` prefix.

`[`

Position prefix. Must be followed by an expression that evaluates to a number (for example `[4` or `[(!pos+3)`) or by a variable having a question mark as in `[?pos`. In the first case, the pattern cannot succeed unless the element following the `[` element is at the indicated position. The `[` element itself does not occupy a position; it sits in front of the indicated position. The second form is for querying the current position. Position 0 is the start of the subject. Positive positions count from the beginning of the subject, negative positions from the end. Position -1 is the position following the last element. (When combined with `%` the meaning is different.)

The above prefixes may be combined. The ordering in which they are input by the user is irrelevant; Bracmat keeps prefixes in this order:

```
[ ~ / # < > % @ ‘ ? ! !!
```

Repeating prefixes in front of the same atom does not convey a new meaning to the pattern, except for the `!` and the `~` prefixes. More than one `!` is interpreted as the `!!` prefix. An odd number of `~` is treated as a single `~`, an even number thereof is treated as none. A `~` in front of other prefixes negates the first of them. The most useful combinations are:

`?!`

in front of an atom causes the atom to be expanded to its binding. This binding is treated as a variable name.

`?!!`

is like `?!`, but expands two levels deep (with an evaluation of the first level expansion), instead of one.

`<>`

is like a solitary `~`.

`/<>5/6`

unifies only with non-integer rational numbers unequal to 5/6.

`~<`

means "greater or equal" ("not less.")

`~>`

means "less or equal."

`~<>`

means "not different," i.e. "the same, in some sense." Strings are compared case insensitive. This applies to the full Unicode table, but defaults to ASCII and the upper 128 characters in the ISO 8859-1 (Latin 1) character set if the characters are not UTF-8 encoded. Subject and pattern can have different encodings and still match with success.

`~#`

does not unify with rational numbers.

`~/`

does not unify with non-integer rational numbers.

`~/#`

unifies only with integer numbers.

`~/#<9`

unifies only with integer numbers less than 9.

`~/#<>0`

unifies only with non-zero integer numbers.

`~@`

unifies only with non-atomic expressions.

`~‘`

backtrack immediately.

`[%` The current subject is stored in the variable `sjt` and the expression carrying this prefix combination is evaluated. If the evaluation succeeds, the match succeeds and vice versa. The subject can be a neutral element of the subject list.

Many of these combinations can be combined further, e.g. `~/#?!!` accepts only an integer number and binds it to the indirect binding of the atom following the prefixes.

If you want to match pattern `!pat` one or more times (this is often written as `{pat}+`), use the complex pattern `(? !pat|‘)`. Likewise, if you want to match `!pat` zero or more times (`{pat}*`), use `(|? !pat|‘)`. These patterns should not be the last subpattern or precede a subpattern that is static and fixes the end point of the repeating sequence, because the correct working of the repeating patterns depends on repeated backtracking

from following subpatterns. Bracmat may be optimized to skip such backtracking and jump to the "right" end position if that is fixed by the next subpattern. In the last resort, you can add a pattern like `()` or `(&)` or `(|)` or `(:)`, which match with an empty list only (assuming that the connecting nodes are spaces, otherwise use 0 in the case of a sum and 1 in the case of a product). Such patterns don't fix the next position. Example:

```
{?} a a a c c:(? a|‘) (|? b|‘) (? c|‘) (&) { {a}+ {b}* {c}+ }
```

The following expression succeeds, because the subpattern doesn't confront the substring `aaak`.

```
{?} @(aaakamcccc:(? a|‘) m (|? b|‘) (? c|‘)) (&)
```

An empty string before the `m` has the effect that Bracmat doesn't optimize the backtracking process away.

```
{?} @(aaakamcccc:(? a|‘) () m (|? b|‘) (? c|‘)) (&)
```

7.3 Minus sign

The minus sign `-` has only its normal arithmetic meaning when used as an unary operator in front of a rational number or the imaginary number `i`.

If a product contains both a rational number and the number `i`, the `i` takes precedence in accepting a minus sign:

```
-7*i*a
```

is evaluated to `7*-i*a`.

The advantage of having both `i` and `-i` becomes clear by considering the following:

```
(-1*i)^1/3
```

evaluates to `(-i)^1/3`, which is written as `-i^1/3`. As expected, this is the complex conjugate of

```
i^1/3.
```

If Bracmat did not have a separate representation for `-i`, then

```
(-1*i)^1/3
```

would evaluate to `i`, (because `i^3` is equal to `-i`),which means that Bracmat would not consider `(-1*i)^1/3` and `i^1/3` as complex conjugates.

The transcendental numbers `e` and `pi` do not accept arithmetic minus signs.

8 String or atoms

A string in Bracmat is the same as an "atom." If you envisage a Bracmat expression as a tree like structure, atoms or strings are to be found in the leafs. In Bracmat terminology, an empty leaf is syntactically represented by *nil*. *nil* is not an atom proper, but an *atom-or-nil*. So not every leaf contains an atom. On the other hand, leafs may contain other things besides *atoms*, such as prefixes.

In Bracmat, atoms are less accessible than trees. Therefore there are some ways to convert atoms to trees and back.

1. Conversion between an atom and its constituent characters:
   - `get$(`*atom*`,MEM,VAP)` puts every character in the literal *atom* in its own leaf in a tree, which has space operators in every node.
   - `str$`*tree* does more or less the inverse.
2. Conversion between an atom and executable Bracmat code:
   - `get$(`*atom*`,MEM)` literally "reads" an atom as though it is a file with Bracmat expressions.
   - `lst$(`*variable*`,MEM)` "writes" the expression that is bound to *variable* to an atom.

Atoms can be used as names for variables, functions, files, etc... Often they are used as literals, such as mathematical symbols or text.

Atoms consist of any number of non-zero bytes, up to the limits set by the operating system and hardware. Atoms can be surrounded by quotation marks, but are in many cases optional. You do need them if you want parentheses, braces, semicolons, operators or prefixes to be part of an atom. All UTF-8 encoded Unicode characters can be used in strings. Some special characters have to be preceded by a backslash:

`\a`
: attention (bell)

`\b`
: backspace

`\t`
: tab

`\n`
: new line

`\v`
: vertical tab

`\f`
: form feed

`\r`
: carriage return

`\\`
: backslash

`\"`
: double quote

If you precede a string with the prefix `@`, then backslashes are treated as normal characters, e.g. `sys$@"C:\dos\edit"`. Instead of the tab and new line characters above, you may enter tabs and new lines by pressing the tab and the return key, respectively.

Examples:

```
{?} this is a "tree" with\nsix leafs
{?} (this is a "tree" with
      seven leafs)
{?} "this" has 4 characters and "" (nil) none
{?} "this is an \"atom\" with 36 characters"
{?} "this string\nno verb"
{?} "this string
no verb either"
{?} "if zero equals one, someone divided by zero" = "1:0&get$(\")y\",MEM)"
{?} get$(!"if zero equals one, someone divided by zero",MEM)
```

9 Symbols

9.1 Literals

In Bracmat, symbols have only literal meaning, unless we explicitly state that we want a symbol to behave like a programming variable. Contrary to most computer languages, Bracmat evaluates an expression with literals not by expanding these literals to their associated values (if they have any) and computing with these values until a result is obtained, but by rearranging and transforming the expression until a stable form is reached.

```
{?} a + a
{!} 2*a
{?} i*i
{!} -1
{?} e^(19/2*pi*i)
{!} -i
```

In Bracmat, the context of a symbol decides whether it is treated as a variable or as a literal. So it is not necessary to kill a variable in order to use its symbol as a literal, the two uses live peacefully together.

```
{?} i=2          { variable i is bound to the literal 2 }
{?} !i^2         { the associated value of i is squared }
{!} 4
{?} i^2          { the literal i (a special one, like pi and e) is squared }
{!} -1
```

```
{?} 7 = prime { the variable 7 is bound to the literal prime }
{?} 7 is !7   { the symbol 7 is used as both a literal and a variable }
{!} 7 is prime
```

9.2 Variables

Variables are represented by *atoms*, but not all *atoms* are variables. The context of a symbol determines whether it is a variable or not:

1. the left operand of the `=` operator, unless this operand has zero length
2. the atom following the `!` and `!!` prefixes
3. within a pattern, a non-zero length atom following the `?` prefix
4. the left operand of the `$` and `'` operators
5. the right operand of the `$` operator in macro constructs (e.g. `'(1+$a)`.)

10 The four evaluators

In Bracmat, a binary operator may have four different effects, depending on the context of the operator. For each of these contexts there is one evaluator. Of these four evaluators, the macro evaluator is relatively unimportant. The four evaluators are:

1. the expression evaluator, which takes care for the transformations of expressions,
2. the *match evaluator*, which handles the unification of pattern expressions with subject expressions,
3. the *macro evaluator*, which merely substitutes certain parts of an expression,
4. the *archivist*, which doesn't do anything but keeping expressions alive.

The expression evaluator is the first evaluator that a newly input expression is confronted with. If necessary, it delegates tasks to one of the other three evaluators. The match evaluator can only delegate tasks to the expression evaluator and to the archivist. The macro evaluator can only delegate tasks to the expression evaluator. The archivist doesn't delegate any tasks to other evaluators.

The cross link is in most cases a binary operator. The exceptions to this rule are in the context of the match evaluator: some (combinations of) prefixes involve the expansion of a chain of variable bindings and all but the last subexpansion demand the expression evaluator. In the scheme below, you'll find the current evaluator in the left column and the successor evaluators in the top row. A cross link is represented by the relevant operator or prefixes. If the change of evaluator only applies to the left (right) operand of the cross link operator, the symbol "l" ("r") is used. If the transition depends on the left operand being *nil*, the symbol "n" is used.

	expression	match	macro	archivist
expression		:	n'r	=r, 'r
match	&r, $, 'l, ?!, !!		n'r	=r, 'r
macro	$r			
archivist				

11 Programming advice

11.1 Debugging

If a program written in the Bracmat language doesn't work properly, the same debugging protocol applies as with other programming languages:

- Test extensively, above all with absurd and trivial input, in order to locate the pain in the many limbs of your program code.
- Create watch points by inserting `out$` instructions at sensible places (entry and exit points of functions, branches, before and after assignments.)
- If you are in doubt whether Bracmat has interpreted your program in the way you intended, use `lst$(`*function-name*`,`*file-name*`)` and inspect the code that is output into *file-name*.
- Errors that are easily made are:
  - Forget that Bracmat may see neutral elements (0 terms, 1 factors, zero length words) at places where this is not what you intend. Are all `%`, ` ‘ ` and `@` prefixes in place?
  - Forget the grouping of operators. For example, `a b c : ?%x ?%y` is grouped as `(a b c):(?%x ?%y)`, but `a,b,c : ?%x,?%y` is grouped as `a,b,(c:?%x),?%y`. Remember that `=` and `.` have very low priorities, often making a pair of parentheses necessary.
  - A misconceived idea about the Bracmat's backtracking mechanism. Unlike other languages with backtracking capabilities, Bracmat does not offer suspend/resume cycles. An expression embodying alternatives does not successively produce each alternative on every evaluation.

11.2 Using `out$` as debugging aid

The best aid in finding out what a program does, is using the `out$` function. The following code is part of a function that computes $n!$.

```
(loop = !k+1 : ?k            { increment k }
             : <!n           { compare (old) k+1 with n; if not less, stop }
      & !fac*!k : ?fac       { multiply fac by k }
      & !loop)               { repeat until k = n }
```

Outside patterns `out$` is most easily used. Inside patterns, if you want to inspect a variable that has just been assigned a new value, you use the `&` operator to temporarily escape into the non-pattern world. If you want to add extra text to the output, remember that the argument to `out$` is returned.

```
(loop = out$!k+1 : (?k & out$(k is !k)) { show k before and after increment }
                 : <!n                  { but before comparison with n }
      & out$("new fac is:" (!fac*!k:?fac)) { show fac after computation }
      & out$(still need !n+-1*!k loops) { you don't always need quotation marks }
      & !loop)
```

Now an example that is faulty. The purpose is to find two equal words in a sentence with a non-linear pattern. This expression succeeds, but finds nothing:

```
(De kok snijdt recht en de meid snijdt scheef
    : (? ?a ? !a ?)
    & out$(!a is occurring twice)
)
```

Check what is unified with `? ?a`. To do so, put a variable after the first `?` and insert an output action after each subpattern.

```
(De kok snijdt recht en de meid snijdt scheef
    : ((?x & out$(x is !x))   { output x after unification }
       (?a & out$(a is !a))   { output a after unification }
       ? !a ?)                { the remainder of the pattern }
    & out$(!a is occurring twice)
)
```

The program would have to backtrack several times until `?a` was unified with `snijdt`, but the match succeeds with `?a` unified with the omnipresent zero length word. A `%` sign avoids this. A back quote ‘ helps speeding up, since it avoids multi-word assignments and forces immediate backtracking.

```
(De kok snijdt recht en de meid snijdt scheef
   : ((?x & out$(x is !x))   { watch the number of words in ?x grow... }
      (%‘?a & out$(a is !a)) { while ?a moves towards snijdt }
      ? !a ?)                { there backtracking stops }
   & out$(!a is occurring twice) { and the message is output }
)
```

11.3 Using dbg’ as debugging aid

Some programming errors may be found with the built-in `dbg` function. The argument of the `dbg` function is evaluated with an internal debugging flag set. With this flag set, suspicious code is warned against.

It is important that the argument is not evaluated before being passed to the `dbg` function.

12 Functions

12.1 Definition of a function

function-name=*var1* [,*var2*, ...].*function-body*

var1, *var2*, etc. are explicitly declared local variables. A function is called by *function-name*$*argument-expression* or *function-name*'*argument-expression*, depending on whether argument expression must be evaluated ($) or not (').

All Bracmat functions have arity one or two, that is, they accept one or two arguments. The *argument expression* to the right of the $ or ' is always represented in the body of the function by a local variable arg.

The returned value of a function is simply the function body after it has been evaluated.

```
{?} square=.!arg^2          { definition }
{?} square$5                { call }
{!} 25
{?} (swap = a,b             { declare local variables a and b }
        .   (!arg:(?a,?b)) { dissect arg to find the real arguments }
          & (!b,!a))        { swap and return }
{?} swap$(I think,I guess)
{!} I guess,I think
```

In a match context, a function call creates a second local variable, sjt (think SuJeT, SubJecT), the current subject. The value returned from a function in a match context is interpreted as a pattern by the match evaluator. However, if the function call fails, the pattern match operation is not attempted and fails as well. The behaviour is not defined if the returned value is negated.

```
{?} ( like
    =
      .   sim$(!arg,!sjt):>9/10 & ?
        |   den$(sim$(!sjt,)):~<(den$(sim$(!arg,0)))
          & ~‘
    )
{?} @( "Dogs and Cats are my enemies": ? like$cat ?)
```

Local variables in Bracmat are shallowly bound dynamically scoped variables. This means that variables that are used in a function but not locally declared in that function, are inherited from the (function or global) context from which the function is called, which in turn may inherit any undeclared variables from another calling context. This scheme contrasts with most programming languages. It is efficient, but the effect of forgetting to declare a local variable can be unexpected behaviour of conceptually unrelated code.

It is possible to declare a function inside another function. Always declare the name of an embedded function as a local variable.

12.2 Lambda calculus, currying

The lambda abstraction

```
(λx.x)y
```

translates to

```
/('(x.$x))$y
```

Bracmat's implementation of lambda calculus is a variant of Bracmat's macro substitution.

The expression

```
/('(x.$x))
```

evaluates to itself, not to something like

```
/(=(x.foo))
```

(assuming that the variable `x` had the value `foo`).

In contrast, the same expression without the leading slash

```
('(x.$x))
```

evaluates to

```
=x.foo
```

The RHS of the `$` operator must be an atom.

In an lambda abstraction

```
/('(x.$x,$y))
```

`$x` is a bound variable and `$y` is a free variable.

The expression `$x` is only replaced by a value if `x` is the variable in the lambda abstraction or a lambda abstraction that contains the lambda abstraction, as in

```
/('(x.(/('(y.($x) ($y)))$aap)))$noot
```

which evaluates to

```
noot aap
```

No Bracmat variables come into play, not even `arg`. Thus, in the example above the value `aap` is bound in `($x)`, but never assigned to a variable `x`.

The expression

```
/('(x.(/('(x.($x) ($x)))$aap)))$noot
```

evaluates to

```
aap aap
```

12.3 Built-in functions

index`$`*array-name*

Both *array-name* and *index* should evaluate to atoms. *array-name* may be preceded by prefixes, such as `?` or `!`. Indexing starts at 0 and is done modulo(size-of-array). Negative values count from the upper end of the array. The chosen index remains in force until a new indexing function is evaluated.

```
{?} tbl$(array,4)              { declare array[0..3] }
{?} a-value : 2$?array         { array[2] := a-value }
{?} array = another-value      { array[2] := another-value }
{?} !array : -1$?array         { array[3] := another-value }
{?} 45 : 1$?array              { array[1] := 45 }
{?} 2'!array : 3'!array        { are array[2] and array[3] equal?
                                 notice use of ' instead of $ }
```

`alc$`*number-of-bytes*

This function allocates memory and returns the starting address of the allocated memory, but crashes the program if not enough memory can be allocated. Access to memory that has been allocated in this way is by means of `pee$` and `pok$`. Any allocated memory should at some time be returned to the memory heap with the function `fre$`.

`arg$` or `arg$`*number*

If Bracmat is started with any arguments (argc > 1) every argument is evaluated from left to right, unless arguments are consumed by calls to `arg$`.

For example

```
bracmat get$myprog -i c:\documents\input.txt -o d:\html\index.html
```

would evaluate

```
get$myprog
-i
c:\documents\input.txt
-o
d:\html\index.html
```

in that order. Evaluating the last four arguments is not very meaningful, however: the backslashes are interpreted as escapes, which they are not. Moreover, the colon and the dot are interpreted as operators. However, the bracmat program `myprog` can call the function `arg$` four times and in that way empty the queue of arguments. `arg$` returns an atom containing an exact copy of the next program argument. Using string matching the arguments can be parsed, if necessary.

Precautions must be taken if the path or name of the bracmat program contains characters

that can be mistakenly interpreted, e.g. (in Windows)

```
bracmat "get$@\"c:Program Files\yourprog.bra\""
```

In *N?X the apostrophes surrounding the first argument must be replaced by quotes.

A second form is `arg$`N, where $0 \leq N <$ argc. `arg$0` will normally return the command name `bracmat` or a path leading to `bracmat`. Here is a simple program `myprog` that demonstrates the two ways of calling the `arg$` function

```
{ myprog }

(test=
  0:?N
&   whl
  ' ( arg$:?argument
    & out$(The next argument is !argument)
    )
&   whl
  ' ( arg$!N:?argument
    & out$(Argument !N is !argument)
    & 1+!N:?N
    )
);

!test;
```

Now running bracmat with these arguments (example is Windows):

```
bracmat get$myprog -i c:\documents\input.txt -o d:\html\index.html
```

results in the following output being written to the terminal:

```
The next argument is -i
The next argument is c:documentsinput.txt
The next argument is -o
The next argument is d:htmlindex.html
Argument 0 is bin\bracmat
Argument 1 is get$myprog
Argument 2 is -i
Argument 3 is c:documentsinput.txt
Argument 4 is -o
Argument 5 is d:htmlindex.html
```

asc$*character*

asc$ returns the integer value that corresponds to the character according to the current table used by the operating system (e.g. an extended ASCII table.)

```
{?} asc$"+" { return ASCII value of character + }
{!} 43
```

bez$

bez$ returns two numbers separated by a dot operator. The first number is the number of currently allocated binary nodes and leafs. The second number is the maximum number of allocated binary nodes and leafs in this session, until now. These numbers are not of much use, except for controlling that there are no memory leaks. The function bez$ can be turned off by setting TELMAX to zero in bracmat.c before compilation.

In interactive mode, with TELMAX set to one, Bracmat shows (1) whether the evaluation was successful or not, (2) the amount of CPU time and (3) the result of calling bez$ after each non-empty evaluation result.

```
{?} 2^10000+-1*3^100000*5^-10000
{!} -133497141423040146945891439048978229224524850760627735534562799398412741460
...218275586967638393602754437083284053816578662576830538455396890640258789062
   S   0,72 sec  (3990.13910)
```

chr$*value*

chr$ returns the character at location value in the current table of characters used by the operating system (e.g. an extended ASCII table.) chr$ fails if *value* equals 0.

```
{?} chr$255 { return the last character from the current table of characters
              (assuming a machine with 8-bit characters) }
{?} ( tolower=.
        !arg:~<A:~>Z                    { if arg is in the range A-Z }
      & chr$(asc$!arg+-1*asc$A+asc$a)   { then return its lower case equivalent }
    | !arg                              { else return arg unchanged }
    ) { works only correctly if the "distance" between lower and upper case
        versions is the same for all characters in the range A-Z }
{?} tolower$G
```

clk$

clk$ returns the number of CPU seconds that that has been spend on running the current session of Bracmat. The number is an unreduced quotient of number of clock ticks and the number of clock ticks per second.

```
{?} clk$
{!} 30375/1000
```

`d2x$`*decimal-value*

`d2x$` converts a decimal number between 0 and 4294967295 ($2^{32}-1$) to an hexadecimal number consisting of characters `0`–`9` and `A`–`F`. On 64-bit platforms the upper bound is 18446744073709551615 ($2^{64}-1$). The function fails if the argument is not an integer number or if the number is outside this range.

```
{?} d2x$(2^32+-1)
{!} FFFFFFFF
```

`dbg'`*expression*

Create warnings in situations that probably are programming errors. Currently, a warning is generated when a function definition can not be found.

```
{?} dbg'(foo$a)
{?} dbg'((myclass.yourfunc)$X)
```

`den$`*rational-number*

The denominator of *rational-number* is returned. There is no built-in numerator extractor function.

```
{?} den$22/7
{!} 7
{?} num = .!arg*den$!arg { home-made numerator function }
{?} num$22/7
{!} 22
{?} den$sim$(,monkey)      { return length of word monkey }
{!} 6
```

`div$(`*rational-number*`,`*rational-number*`)`

`div$` returns the (integral) quotient of its arguments.

```
{?} div$(123/45,67/890)
{!} 36
```

`fil$(`[[*file-name*][`,`*option*[`,`*number*[`,`*value-to-output*]]]]`)`

`fil$` is a multi-purpose low level I/O function.

`fil$(`[*file-name*]`,`*mode*`)`

set file mode, open a file in a file mode

`fil$(`[*file-name*]`,`*type*`,`*size*[`,`*number*]`)`
: prepare for reading or writing fixed sized records

`fil$(`[*file-name*]`,STR`[`,`*stop*]`)`
: prepare for reading or writing variable sized record

`fil$(`*file-name*`,TEL)`
: tell position inside file

`fil$(`*file-name*`,`*whence*`,`*offset*`)`
: go to file position

`fil$(`[*file-name*][`,,`*number*]`)`
: read from file

`fil$(`[*file-name*]`,,`*number*`,`*value*`)`
: write to file

`fil$(`[*file-name*]`,SET,-1)`
: close a file

`fil$(`[*file-name*]`,`*mode*`)`

Set file mode, open file in file mode.

Option *mode* is one of the following:

`r`
: open text file for reading

`w`
: create text file for writing, or truncate to zero length

`a`
: append; open text file or create for writing at EOF

`rb`
: open binary file for reading

`wb`
: create binary file for writing, or truncate to zero length

`ab`
: append; open binary file or create for writing at EOF

`"r+"`
: open text file for update (reading and writing)

`"w+"`
: create text file for update, or truncate to zero length

`"a+"`
: append; open text file or create for update, writing at EOF

`"r+b"` or `"rb+"`
: open binary file for update (reading and writing)

`"w+b"` or `"wb+"`
: create binary file for update, or truncate to zero length

`"a+b"` or `"ab+"`
: append; open binary file or create for update, writing at EOF

`fil$(`[*file-name*]`,`*type*`,`*size*[`,`*number*]`)`

Prepare for reading or writing fixed sized records.

Option *type* is one of the following:

`CHR`
: character or string I/O

`DEC`
: number I/O

Option *size* must be a non-negative integer and determines the number of bytes that are read or written as one chunk during a future call to `fil$`. If type is `DEC`, only values 1, 2 and 4 are valid, corresponding to 1, 2 and 4 byte sized integers, respectively. Notice that 2 and 4 byte integers are not portable between implementations with different byte order. The optional *number* tells how many read or write operations of size *size* have to be performed. If type is `DEC`, the product of *size* and *number* may not be greater than 4. If a number is read or written in 2 or more chunks, the least significant bytes or 16 bit words are read or written first (little-endian.)

`fil$(`[*file-name*]`,STR[`,*stop*]`)`

Prepare for reading or writing variable sized record.

Option *stop* is a string of characters. If the read character or the character to be written is equal to one of the characters in the *stop* string, reading or writing stops. The default is not to stop until the end of the file (reading) or the end of the string (writing.)

`fil$(`*file-name*`,TEL)`

Tell the position inside the file.

Returns the current value of the file position indicator.

`fil$(`*file-ame*`,`*whence*`,`*offset*`)`

Go to file position.

Sets the current value of the file position indicator to an *offset* based on the value of *whence*.

Option *whence* is one of the following:

`SET`
: start of file

`CUR`
: current file position

`END`
: end of file (in some implementations, binary files may not handle `END`)

For a text file, offset must be 0, or the value returned from a call to `fil$(`*file-name*`,TEL)`, in which case *whence* must be `SET`.

`fil$(`[*file-name*][`,,`*number*]`)`

Read from a file.

Reads (*mode* permitting) *number* chunks of *size* bytes. When reading variable sized records (`STR`), `fil$` returns a dot-separated list of two elements: the found stop character and the read string (which does not contain the stop character.)

```
{?} fil$("mytext.txt","rb")       { open for reading in binary mode }
{?} fil$(,STR)                    { prepare for reading until the next 0-byte }
{?} fil$:(?line.?stop)            { read until the next 0-byte }
{?} fil$(,SET,-1)                 { close the file }

{?} fil$("mytext.txt","rb")       { open for reading in binary mode }
{?} fil$(,STR,"\n \t\r")          { prepare for reading until the next whitespace }
{?} :?words                       { in a moment, accumulate all words in this variable }
{?} whl'(fil$:(?word.?stop)&!word !words:?words)&   { read all words }
{?} fil$(,SET,-1)                 { close the file }
{?} !words                        { show all words, in reversed order }
```

`fil$(`[*file-name*]`,,`*number*`,`*value*`)`

Write to a file.

Writes (*mode* permitting) *number* chunks of size bytes from *value*. If type is `DEC`, *number*$\times$*size* must be 1, 2, 3 or 4. *value* must be an integer value and is cast to a binary number with at most *number*$\times$*size*$\times$8 *bits*. This number is stored in *number*$\times$*size* bytes, which in turn are output. If number is greater than 1, the byte(s) with the least significant digits are output first. In machines with little-endian byte-order, only the product *number*$\times$*size* matters. If *type* is `CHR` and the length of *value* is shorter than *number* $\times$ *size*, *value* is padded with spaces (to the right.) If type is `STR`, *number* must be the empty string.

`fil$(`[*file-name*]`,SET,-1)`

Close a file.

An open file is closed by specifying an impossible file position.

`flg$(=`*expression*`)`

`flg` returns a copy of the expression without prefixes ("flags") and a new leaf with the prefixes of the original expression. These two results are coupled with a dot operator, the prefixes to the left and the expression without prefixes to the right. The result is protected against evaluation by a `=` operator.

```
{?} flg$(=~#<>?%@a)
{!} (=~#<>%@?).a
```

Use macro evaluation if the expression to be split is the value of a variable:

```
{?} X=~(%+%)
{!} X
{?} flg$('$X):(=?prefixes.?expr)
{!} =~.%+%
{?} glf$('($prefixes.%*%))
{!} =~(%*%)
```

`glf$(=`*prefixes*`.`*expression*`)`

`glf` returns a copy of *expression* with *prefixes* added to its prefixes. If *expression* has one or more prefixes also present in *prefixes*, then `glf` fails. Therefore, `glf` can be used to test for the presence of one or more prefixes. The result is protected against evaluation by a `=` operator. The function `glf` has an effect that is the opposite of `flg`.

```
{?} glf$flg$(=?a)
{!} =?a
```

Use macro evaluation if the expression to be split is the value of a variable.

```
{?} X=?!x
{!} X
{?} flg$('$X):(=?prefixes.?expr)
{!} =?!.x
{?} glf$('($prefixes.z))
{!} =?!z
```

`fre$`*memory-address*

`fre$` returns a chunk of memory to the memory pool (heap). The only valid parameter is a return value of `alc$`. Applying `fre$` to a chunk of memory that was never allocated or that has been returned already results in undefined behaviour of the program.

```
{?} alc$1000:?p   { allocate chunk of 1000 bytes from the memory pool }
{?} fre$!p        { return this chunk to the memory pool }
```

get$(*atom-or-nil*[,MEM][,ECH][,VAP][,STR][,TXT|BIN][,JSN]|[[,X]|[,HT],ML[,TRM]])

get$ reads and interprets characters in a string (internal memory) or file (external memory or keyboard).

Options:

MEM

Present: The name of the first parameter is the source of the characters (MEMory.) Not present: A file with the name of the first parameter is the source.

ECH

Present: The characters are echoed to the screen as they are read. Not present: No echo.

VAP

Present: The (8-bit) characters are read as is. Extra spaces are inserted between the characters (VAPorised.) Not present: No extra spaces are added.

STR

Present: The characters are read into one string. Not present: The characters are interpreted as parts of a Bracmat expression and evaluated after the whole expression has been read.

TXT

Present: File is read in "text" mode (file mode r.) Not present: File is read in binary mode (file mode rb).

BIN

Present: File is read in binary mode. Not present: File is still read in binary mode.

JSN

Present: The input is parsed as JSON.

ML

Present: The input is parsed as markup (SGML, XML or HTML). Any unrecognised entity reference is preceded by a DEL character (ASCII code 127 (decimal)). Any DEL character in the input is also preceded by a DEL character, similar to how the traditional escape character \ is itself represented by doubling it: \\.

HT (together with ML)

Present: HTML entities are translated to UTF-8 characters. Not present: Only XML entities are translated to UTF-8 characters.

X (together with ML)

Present: Only XML entities are translated to UTF-8 characters. Processing instructions are assumed to have the syntax ?...?. Not present: Processing instructions are assumed to have the syntax ?... .

`TRM` (together with `ML`)

Present: Heading and trailing whitespace is cut down to a single space in character data. Not present: All whitespace is kept. Input can be exactly reproduced when writing, except HTML entities.

The `VAP` option is evaluated before the `STR` option.

Applications:

```
get'(matrix,ECH)
```

Read file `matrix` and evaluate the expressions (delimited by semicolons) therein. If the system finds a syntax error in a multiple expression file, the `ECH` option makes it easier to locate the error.

```
get'(matrix,STR):?intern
```

Read file `matrix` into a string called `intern`. If this file contains Bracmat instructions, they hereafter exist in a sleeping state in memory.

```
get$(!intern,MEM)
```

`!intern` is, in this example, expanded to an atom with a very long name, namely all of the text of file `matrix`. The sleeping expressions are evaluated one after the other, just as if `get'matrix` was evaluated.

```
get'(,VAP):?space-list
```

Read characters from standard input (normally keyboard) until next line feed character. Put each character into an atom. Put all atoms into a linear list with space operators. Bind this list to the name `space-list`.

```
get'(")y",MEM)
```

Read the sleeping expression `)y` from memory. The lexical scanner will find an unbalanced right parenthesis, which could mean that this Bracmat session should stop. The `y` confirms this assumption and the program will come to an end immediately. If the `y` hadn't been present, Bracmat would ask `end session? (y/n)` after which the user has to choose. This trick is useful in batch processing.

If the first parameter is *nil* or `stdin` and the `MEM` option is not used, input is coming from standard input. Take care for putting filenames in double quotes if they contain any characters that can be misunderstood, such as dots, (back)slashes or dollars.

`lst$(`(=*expression*) or *variable**[`,LIN`][`,RAW`]`)`

`lst$` outputs all present bindings of one or more variables to standard output. If a variable is local to a function and a variable of the same name already exists in the context of another function invocation or globally, all instances will be listed!

If a variable is listed to stdout, the listing starts with `(varname=` and ends with `);`. The surrounding parentheses are suppressed if a variable is listed to a file.

Expressions headed by the `=` operator can be used instead of variable names. In that case no semicolon is appended. The listing starts with `(=` and ends with `)`, followed by a newline.

With the `RAW` option, only the expression right of the `=` operator is listed.

The `LIN` option suppresses most newlines and all other unneeded whitespace characters.

If the first parameter is the zero-length string, then all variables with names starting with a character below ASCII 128 are shown. If a variable has more than one binding (arrays/stacks) then the current value is preceded by a `>` sign. The second parameter is optional.

```
{?} lst$help        { shows this programme on screen,
                      unless stdout has been redirected }
{?} lst$(tay,LIN)  { listing without indentations of function tay$ }
```

If `LIN` is not present, output is very much indented, sometimes making it more readable for humans. If `LIN` is present, output is as compact as possible.

`lst$(`(=*expression*) or *variable**`,MEM[,LIN][,RAW])`

The difference with the preceding form is that output takes place to memory. What is normally visible on screen is put in one atom, which is the return value of the call to `lst`. It has the opposite effect of `get$(`*atom*`,MEM)`. In theory one can save some memory by writing an expression to a single leaf node, because each node in an expression has some memory overhead. In practice the saving can be quite small or non-existent, because Bracmat expressions often share part of their data structures with other expressions.

Example:

```
{?} lst$(fct,MEM):?sleeping-fct
{?} tbl$(fct,0)    { remove function fct$ from memory }
{?} { ... celebrate space-saving, until fct$ is needed... }
{?} get$(!sleeping-fct,MEM)
```

`lst$(`(=*expression*) or *variable**`,`*file-name*`,NEW|APP[,LIN][,TXT][,BIN])`

This time, output is sent to the named file instead of standard output. The third argument is explained below. Code that has been saved with `lst$` can be reloaded with `get$`.

```
{?} lst$(,"all",NEW,LIN)
    { write all current code without indentations to file all }
{?} lst$(tay,taylor,NEW)
    { save function tay$ to file taylor in indented format }
```

One of the options `NEW` and `APP` must be present:

`NEW`
: Tells the computer to open a new file or overwrite an old one.

`APP`

Directs output to an existing file. If the file does not exist, it is created first.

Options `TXT` and `BIN` overrule the default file mode of the functions `put` and `lst`.

`TXT`

Tells Bracmat to write a file in text mode. This is the default file mode for `put`. In Windows, each line feed character will be preceded by a carriage return. In Unix and Linux, text mode and binary mode are the same.

`BIN`

Tells Bracmat to write a file in binary mode. This is the default file mode for `lst`.

`map$(`*fnc*`.`*list*`)`

`map$` produces a list containing the results of applying *fnc* to each member in *list*. *fnc* can be the name of a function, a function definition ("anonymous function") or a lambda abstraction. *fnc* can also be the name of a built-in function. (See the first example.) The members in *list* must be separated by the space operator.

```
{?} map$(rev.aap noot mies)
{!} paa toon seim
{?} map$(/('(x.$x^2)).1 2 3 4)
{!} 1 4 9 16
{?} map$((=.!arg^2).1 2 3 4)
{!} 1 4 9 16
{?} (square=.!arg^2)&map$(square.1 2 3 4)
{!} 1 4 9 16
{?} (  map
    $ ( (
          =   x
            .     !arg:?*?^%?x*?*?^!x*?
                & (!arg."same exponent" !x)
              | (!arg.)
             )
            . a*b^3*c^2 d*f^3*g^2*h*j^3 o*p
           )
         )
{!} (a*b^3*c^2.)
    (d*f^3*g^2*h*j^3.same exponent 3)
    (o*p.)
```

`mem$[EXT]`

`mem$` produces a list of all currently existing variables, except those beginning with a character above ASCII 126. The `EXT` option adds information about the number of occurrences (array or stack size-1) of those variables which have more than one occurrence and shows which of them is currently in focus (index into array: 0 .. size-1). The predefined function `cat$` makes use of `mem$`.

```
{?} mem$
{?} mem$EXT
```

`mod$(`*number*`,`*divisor*`)`

`mod$` divides *number* by *divisor*. The remainder is returned.

```
{?} mod$(22,7)
{!} 1
```

`new$`*object* or `new$(`*object*`,`*args*`)`

`new$` creates a shallow copy of an *object* and calls the method `new` of the new object, if there is one. With the second form, *args* is passed to the method `new`.

Example:

```
{?} (patient=
    (name=(first=John),(last=Bull)) { name is a copy, first and last are not }
  , (age=20)
  , (new
    =
      .   out$"hello world"
        & new$(its.name):(=?(its.name)) { create fresh copies of first and last }
        &   !arg
          : (?(its.name.first).?(its.name.last).?(its.age))
  ))
{?} new$(patient,(Albert.Keinstein.42)):?x
{?} new$(patient.name):(?Name)
{?} Alice:?(Name..first)
{?} new$(('$patient),(Albert.Keinstein.42)):?y {x and y are identical !}
{?} new$(=(a=1),(b=2)):?ab {"die" is called when ab is reassigned}
{?} 3:?(ab..a)
{?} new$(=(a=1),(b=2),(new=.),(die=.)):(=?cd) {"die" is called at once!}
{?} 3:?(cd.a)
```

When an object was created with the `new$` function, an internal flag is set in the object telling the system that the `die` method must be called just before deletion of the object.

The `die` method, like the `new` method, is optional and should be used to do clean-up.

`pee$(`*address*`[,`*size*`])`

Depending on *size*, a 1, 2 or 4 byte sized integer allocated at *address* is returned.

2 and 4 byte integers may only start at addresses that are multiples of 2 and 4, respectively. *address* is lowered to the nearest allowable value, if needed. Notice that multi-byte integers are stored differently in little-endian (iAPx86, VAX, ARM) and big-endian (MC680x0) machines. Many operating systems abort programs that try to access non-existent or protected memory areas.

```
{?} chr$pee$34567   { return value at address 34567 (1 byte) as a character }
{?} pee$(34567,2)   { return value at address 34566 (2 bytes) }
{?} pee$(34567,4)   { return value at address 34564 (4 bytes) }
```

`pok$(`*address*`,`*value*`[,`*size*`])`

Depending on *size*, a 1, 2 or 4 byte sized integer is stored at address.

2 and 4 byte integers may only start at addresses that are multiples of 2 and 4, respectively. *address* is lowered to the nearest allowable value, if needed. Notice that multi-byte integers are stored differently in little-endian (iAPx86, VAX, ARM) and big-endian (MC680x0) machines. Many operating systems abort programs that try to access non-existent or protected memory areas.

```
{?} pok$(34567,asc$K) { store the internal value of the character K at
                        memory location 34567 (as 1 byte) }
{?} pok$(34567,-1,4)  { store 2^32-1 at memory location 34564
                        assuming 1-complement arithmetic }
```

`put$(`*expression*`[,LIN])`

Sends *expression* to standard output. The cursor is positioned after the last output character. The predefined function `out$` does the same as `put$`, with the exception that it positions the cursor on the beginning of the next line.

```
{?} put$("b+a" is after evaluation b+a)
```

`put$(`*expression*`,MEM[,LIN])`

expression is stringified and placed into an atom, which is the return value of the call. This use of `put$` is similar to the `str$` function, but whereas `str$` suppresses the space operator, `put$` transfers every character, including space operators, to the atom.

`put$(`*expression*`,`*file-name*`,NEW|APP[,LIN])`

Write *expression* to the named file.

```
{?} put$(tay$(e^x,x,10),"e.out",APP)
```

`ren$(`*oldname*`,`*newname*`)`

Renames a file or directory or moves a file. The `ren$` function succeeds unless a syntactic error was made. If there is an error at the operating system level, one of the following codes is returned:

`EACCES`
: File or directory specified by *newname* already exists or could not be created (invalid path); or *oldname* is a directory and *newname* specifies a different path.

`ENOENT`
: File or path specified by *oldname* not found.

`EINVAL`
: Name contains invalid characters.

If the command succeeds at the operating system level, the value 0 is returned.

`rmv$`*file-name*

Removes a file. The `rmv$` function succeeds, unless a syntactic error was made. If there is an error at the operating system level, one of the following codes is returned:

`EACCES`
: Indicates that the path specifies a read-only file or that the file is open.

`ENOENT`
: Indicates that the filename or path was not found or that the path specifies a directory.

`EINVAL`
: Name contains invalid characters.

If the command succeeds at the operating system level, the value 0 is returned.

`rev$`*atom*

Reverses the order of bytes in an atom. This function can be useful in case of a string match that asks for the last occurrence of a pattern.

The `rev$` function succeeds on all atoms and fails on all other expressions.

`sim$(`*atom-or-nil*`,`*atom-or-nil*`)`

`sim$` uses the Ratcliff/Obershelp pattern matching algorithm in establishing a measure of the similarity between its two (atomic) arguments. The returned value is an unsimplified fraction. The denominator is the sum of the numbers of characters in both arguments. The numerator is the total number of characters that have been matched successfully. Matching is case-insensitive. This applies to the full Unicode table, but defaults to ASCII and the upper 128 characters in the ISO 8859-1 (Latin 1) character set if the characters are not UTF-8 encoded.

```
{?} sim$(colour,Color)
{!} 10/11
{?} den$sim$(,"this is an easy way to find this string's length")
{!} 48
{?} div$(sim$("similarity rounded","and in procents")+1/200,1/100)
{!} 30
```

`str$`*expression*

`str$` writes *expression* into one single atom. All atoms, prefixes and operators, with the exception of the space operator, are copied to the output string. Main use: pasting of two or more atoms.

`swi$(`*interrupt-number*`.`*input-value*`,[`*input-value*`,...])`

This function is the most operating-system-dependent function in Bracmat. Currently, it is implemented for RISC-OS (Archimedes) and 16-bit MS-DOS versions of Bracmat. All arguments must be integer values. The list of input values (registers r0 and upwards) need not be complete. Missing values are assumed to be zero. Blocks of memory should be passed by allocating memory with `alc$` and passing the returned value. The returned value has the form

`(`*error-code*`.`*output-value*`,`*output-value*`,...)`

An error code of 0 means that no error is reported. In the MS-DOS version, the input registers are AX, BX, CX, DX, BP, SI, DI, DS, ES and FLAGS, respectively.

```
{?} {RISC-OS only}
{?} putstr=(loop,c,buf,ret.          { goal: copy argument to memoryblock }
      alc$(den$sim$(!arg,)+1):?buf:?ret { allocate block for string to fit }
    & get$(!arg,MEM,VAP):?arg        { argument → single characters }
    & (loop = !arg:%?c ?arg & pok$(!buf.asc$!c.1) & !buf+1:?buf & !loop)
    & ~!loop                         { poke each character into memory block }
    & pok$(!buf.0.1)                 { poke string-delimiting zero }
    & !ret)                          { return pointer to block }
{?} putstr$"OS_EvaluateExpression":?inbuf { create pointer to string }
{?} 57:?"OS_SWINumberFromString"     { from manual }
```

```
{?} swi$(!"OS_SWINumberFromString".0,!inbuf):(?error.?nummer,?) { find
      interrupt number corresponding with the string OS_EvaluateExpression }
{?} fre$!inbuf { deallocate block containing copy of input string }

{?} {MS-DOS only}
{?} gotoxy = (VIDEO,setCursorPosition,videoPage0.
{?}   16:?VIDEO                  { interrupt number 10H }
{?} & 2*256:?setCursorPosition   { AH }
{?} & 0*256:?videoPage0          { BH }
{?} & !arg:(?x,?y)               { DL and DH }
{?} & swi$(!VIDEO.!setCursorPosition,!videoPage0,0,!x+256*!y)
{?} )
{?} gotoxy$(0,0) & put$(top left corner)
```

`sys$`*command-line-command*

In most environments, `sys$` passes its argument to the command line interpreter. Therefore, `sys$` has a functionality that very much depends on the operating system in which Bracmat runs. One has to take care for memory limitations and the possibility that `sys$` may never return. Possible uses are for example:

- instructions to the video display unit for cursor movement or graphics
- running another program, such as an editor
- manipulation of files (directory listing, copying, etc.)

The functions `put$(`*expression*`,MEM)` or `str$`*expression* may be used for constructing an argument for `sys$`:

```
{?} file = bracmat.c
{?} !file:((?stem.?)|?stem) { remove file extension and put result in stem }
{?} sys$str$("copy " !file " " !stem.bak)
```

The `sys$` function succeeds, unless a syntactic error was made. If there is an error at the operating system level, one of the following codes is returned:

`E2BIG`
: Argument list (which is system-dependent) is too big.

`ENOENT`
: Command interpreter cannot be found.

`ENOEXEC`
: Command interpreter file has invalid format and is not executable.

`ENOMEM`
: Not enough memory is available to execute command; or available memory has been corrupted; or invalid block exists, indicating that process making call was not allocated properly.

If the command succeeds at the operating system level, the value 0 is returned.

`tbl$(`*variable*`,`*array-size*`)`

The named variable is (re-)sized to an array with *array size* elements. Resizing always affects the elements with the highest indexes first: shrinking means that the last elements are lost, expanding creates new zero-valued elements at the end.

If *array-size* equals zero, the named *variable* ceases to exist in memory. This is the only way in Bracmat to get rid of global variables.

Stacks and arrays are exactly the same thing in Bracmat. Therefore, it is not possible to declare arrays locally.

Bracmat never accesses arrays as a whole; there is always just one element that is in focus. By issuing an instruction of the form *index*`$`*array-name* or *index*`$`*array-name* you can explicitly tell Bracmat to put focus on some element. Bracmat does this automatically in the case of pushing and popping local variables onto and from a stack.

```
{?} tbl$(bigarray,16000)
{?} lst$bigarray            { this may take a long time to execute }
{?} tbl$(bigarray,0)        { remove bigarray }
{?} lst$bigarray
```

`vap$(`*fnc*`.`*string*`)` or `vap$(`*fnc*`.`*string*`.`*separator*`)`

The version without specified separator splits the string in a list of characters and applies `fnc$` to each character. `vap$` checks whether the string is valid UTF-8 before splitting the string. If not, it splits the string in 8-byte characters, even if some substrings could be interpreted as valid UTF-8.

The version with a specified separator splits the string at each occurrence of the separator. The separator can be any non-zero length string. The separator does not become part of the output. This version is especially useful for reading CSV files and tab-separated files. Note that, if the last character of the input string is a separator, the last call to `fnc$` is with a zero-length string. This will often happen if a newline character terminates every line in the input string and the newline character is specified as a separator.

`whl'(`*expression*`)`

`whl'` implements a `while` *expression* loop. The expression is repeatedly evaluated until it fails. `whl'` always succeeds. Notice that `whl'` is faster than loops using tail recursion.

`x2d$`*hexadecimal value*

`x2d$` converts an hexadecimal number between 0 and FFFFFFFF to a decimal number. On 64-bit platforms the upper bound is FFFFFFFFFFFFFFFF. The function fails if the argument is not a string with a length between 1 and 8 only containing the characters `0`–`9`, `a`–`f` and `A`–`F`.

```
{?} x2d$ffffffff
{!} 4294967295
```

12.4 Predefined functions

Besides hard-coded built-in functions, Bracmat offers a number of soft-coded functions which behave as user defined functions in all respects. They are redefinable and removable, for example. Some functions are called by the interpreter itself and should never be changed by the user. Such functions have names that start with an 8-bit character with the high-bit set. Bracmat has been drilled to leave these names out when the user asks for a list of variable names (`lst$` or `mem$`), so you will not notice their existence.

`abs$`*expression* and `sgn$`*expression*

`sgn$` determines the sign of the numerical factor of expression. If the sign is `-`, then `sgn` returns `-1`. In all other cases the returned value is `1`.

`abs$`*expression* is defined as `sgn$`*expression*`*`*expression*.

```
{?} sgn$(-1*i)
{?} abs$(-7*a)
```

`cat$(`[*include-list*][`,`[*exclude-list*][`,EXT`]]`)`

`cat$` is like the built-in function `mem$`, but offers the possibility to exclude names that are not in the first parameter and/or to exclude names that are in the second parameter. The optional third parameter adds information about array size and current index value, e.g. (arg,5,5). If the first parameter is missing, this is taken to mean that *no* names are excluded (unless by virtue of the second parameter.) You can use `cat$` to save the state of the variable space for later use, e.g. removing all variables that have been created since.

```
{?} cat$(,,EXT):?save-state   { mem$EXT is also OK }
{?} newvar1=12345             { create new variables }
{?} tbl$(newvar2,100)
{?} cat$(,!save-state,EXT)    { show the newcomers }
```

`cos$`*expression* and `sin$`*expression*

These functions produce cos(*expression*) and sin(*expression*), expressed in powers of e. In this way, expressions with goniometric functions can be differentiated and, sometimes, simplified.

`fct$`*expression*

`fct$` uses some heuristics in trying to factorise *expression*.

```
{?} 1+(2*a^3+6/7*t)*(3*x+4*y+z^-1)+-1:?sum
{?} fct\d!sum
{!} 6/7*z^-1*(1+3*x*z+4*y*z)*(t+7/3*a^3)
```

`flt$(`*rational-number*`,`*number-of-decimals*`)`

`flt$` converts a rational number to a floating point presentation. The result is stored in an atom. This function is meant for output, Bracmat does not use floating point numbers itself.

```
{?} flt$(123/456,78)
{?} flt$(sub$(tay$(sin$x,x,40).x.11/7),12)
```

`jsn$`*expression*

`jsn$` serializes expression to JSON format.

Suppose `mars.json` contains

```
{ "planet": "Mars",
  "moons": ["Deimos", "Phobos"],
  "Eccentricity": 0.0934 }
```

Read `mars.json` into variable `mars`:

```
{?} get$("mars.json",JSN):?mars
{!}     (Eccentricity.934/10000)
      + (moons.,(.Deimos) (.Phobos))
      + (planet..Mars)
    ,
{?} !mars:(?name-value-pairs,)
{!}     (Eccentricity.934/10000)
      + (moons.,(.Deimos) (.Phobos))
      + (planet..Mars)
    ,
```

Now add the fact that Mars' gravity, compared to the Earth's, is about 3711/9807:

```
{?} !name-value-pairs+(ratio.3711/9807):?name-value-pairs
{!}   (Eccentricity.934/10000)
    + (moons.,(.Deimos) (.Phobos))
    + (planet..Mars)
    + (ratio.3711/9807)

{?} (!name-value-pairs,):?mars
{!}     (Eccentricity.934/10000)
```

```
        + (moons.,(.Deimos) (.Phobos))
        + (planet..Mars)
        + (ratio.3711/9807)
      ,
```

Finally, serialize the new data to JSON

```
{?} jsn$!mars
{!} {"Eccentricity":0.0934,"moons":["Deimos","Phobos"],
     "planet":"Mars","ratio":3.784031814010400734l7E-1}
```

Numbers are, if possible, represented in exact fixed decimal notation. If rounding is inevitable, numbers are represented in floating point notation with 21 digits. Numbers that have been read from JSON input are always considered to be unrounded, and therefore represented in fixed decimal notation upon writing to JSON.

Objects have their elements sorted.

No attempt is made to make the result look pretty using indentation.

Mapping between JSON and Bracmat expressions

A JSON object is represented as a sum of dotted pairs as the LHS (left hand side) of a comma operator, while the RHS must be empty. The original ordering of members may be lost, as the sum operator sorts its terms. In each dotted pair, the LHS represents the object member's key while the RHS represents the object member's value. An empty object is represented as `(0,)`.

A JSON array is a space separated list as the RHS of a comma operator with an empty string as LHS.

A string is mapped to the atomic RHS of a dot operator, leaving the LHS empty.

`true`, `false` and `null` map onto themselves.

`MLencoding$`*internal-expression-of-XML-or-HTML-data*

This function parses the data for indicators of the used character encoding.

`nestML$`*internal-expression-of-XML-or-HTML-data*

After reading XML or HTML data, all text and tags are in a flat list. The function `nestML` attempts to match opening and closing tags and creates a tree structure. This works best for XML data.

`out$`*expression*

Uses the built-in function `put$` to output expression to the output stream (usually the screen.) Output is ended with a new line. Normally, `out$` returns its argument. `out$` is a good debugging tool, but `put$` is slightly safer, as it handles failing arguments in the correct way, contrary to `out$`.

```
{?} put$a & put$b
ab{!} b
{?} out$a & out$b
a
b
{!} b
```

`sub$(`*expression*`.`*pattern*`.`*replacement*`)`

Substitution function. Argument 1: subject. Argument 2: pattern. Argument 3: replacement for subexpressions in subject matched by pattern.

`tay$(`*expression*`,`*variable*`,`*number-of-terms*`)`

A Taylor expansion is applied to *expression*. The second argument is the independent variable. The third argument denotes the number of terms, including vanishing terms.

```
{?} tay$((cos$x)^-1,x,20)
```

`toML$`*internal-expression-of-XML-or-HTML-data*

This function creates a string containing XML or HTML formatted data from an internal representation of those data. `toML` works both with nested and unnested data. All necessary character conversions are made in `PCDATA` and attributes.

`chu$`*value*

`chu$` returns the UTF-8 character at Unicode code point *value*. `chu$` fails if *value* equals 0 or less or if *value* exceeds 2147483647 (7FFFFFFF). According to the UTF-8 standard values above 1114111 (10FFFF) are illegal.

`low$(`*atom-or-nil*`)`

`low$` converts a string to all-lowercase. The characters `A`–`Z` are converted to `a`–`z`. Other characters are handled as UTF-8 encoded Unicode characters, but default to ISO 8859-1 (Latin 1) if the argument is not valid UTF-8.

`upp$(`*atom-or-nil*`)`

`upp$` converts a string to all-uppercase. The characters `a`–`z` are converted to `A`–`Z`. Other characters are handled as UTF-8 encoded Unicode characters, but default to ISO 8859-1 (Latin 1) if the argument is not valid UTF-8.

`utf$` *UTF-8-character*

`utf$` returns the Unicode code point of the UTF-8 character. The function fails if the string is too short or too long, or if the sequence is an invalid UTF-8 string.

It is safe to use `utf$` in a pattern: `@(!txt:(?%c & utf$!c) ?)` If the value of `txt` starts with an valid UTF-8 sequence, Bracmat backtracks until the value of `c` matches the UTF-8 sequence. If `txt` starts with a sequence that is not UTF-8, Bracmat stops backtracking when that fact has been established.

13 Methods

A method is function that is a member of an object. A method definition is not different from a function definition, but the ways in which methods and a functions are called are. While a function call only requires the name of the function, a method call requires the name or definition of an object and the name of the method, separated by a dot operator. In the body of a method it is usually necessary to refer to other members of the same object - that is one of the primary objectives of using objects. Such self-references start with `its`, which is analogous to `this` in C++. Here is an example of an object that represents a rectangle with an `x-size` and a `y-size` for the lengths of two orthogonal sides. Apart from these two data members, the object also has a member function `area$` that returns the product of the `x-size` and the `y-size`, and another member function that computes the length of the diagonal.

```
{?} (rect=
    (x-size=5)
    (y-size=12)
    (area=.!(its.x-size)*!(its.y-size))
    (diagonal=.(!(its.x-size)^2+!(its.y-size)^2)^1/2)
    );
{?} (rect.area)$
{!} 60
{?} (rect.diagonal)$
{!} 13
```

14 Hash tables

If you need to manage a large data set it may be a good idea to use a hash table instead of a list. Storing, retrieving and deleting are costly processes in lists, but cheap in hash tables. Handling hash tables in Bracmat is very simple. You create a hash table as follows

```
{?} new$hash:?myhash
```

Hereafter, `myhash` refers to a hash table and is treated in the same way as a user defined object. Bracmat keeps the load factor between 50 and 100, rehashing as necessary.

If you know the hash table is going to be much bigger than about 100 bins, you can

suggest the size of the table to Bracmat, but this is not necessary:

```
{?} new$(hash,1000000):?mybiggerhash
```

This tells Bracmat to start off with a million bins.

The following methods are defined for hash tables:

`find`
: find all values for a given key

`insert`
: insert a value for a key

`remove`
: remove a key and all its values

`New`
: creates a hash table, cannot be called programmatically

`Die`
: cleans up a hash table, cannot be called programmatically

`ISO`
: make all key access case-insensitive

`casesensitive`
: make all key access case sensitive (default)

`forall`
: apply a function to all key-value pairs

`(myhash..find)$key:(?Key.?Value) ?OtherKeyValuePairs`

Returns a blank-separated list of key-value pairs, all with the same key.

`(myhash..insert)$(Key.Value)`

Inserts the key `Key` with the value `Value`. Multiple values for the same key are possible and the same value can be inserted more than once for the same key.

`(myhash..remove)$key:?KeyValuePairs`

Removes the key with all its values and returns a blank-separated list of key-value pairs.

`New`

This is a method of the hash class that is called by the system when it evaluates `new$hash`. It cannot be called from user code.

`Die`

This method is called when a hash object is deleted. It is not directly called from user code. (Compare with a C++ destructor.) You can add your own clean up code by writing a `die` method and adding it to a hash object once it is created.

Example:

```
{?} new$hash:?myhash;                  { create a hash table myhash  }
{?} ((
    =   ( Insert                       { add a method Insert to myhash that
                                         only allows one value per key }
        =   K,V
          .     !arg:(?K.?V)
              & (Its..find)$!K:(?.?v)
              &   out
                $ ( str
                  $ ( "Key "
                      !K
                      " already present"
                      ( !V:!v&" with same"
                      | ", but with different"
                      )
                      " value "
                      !v
                    )
                )
            | (Its..insert)$!arg
        )
        (die=.out$"Who ordered th") { this method is called just before
                                      the object is deleted }
    )
  : (=?(myhash.)))

{?} (myhash..Insert)$(X.12);           { insert the value 12 for key X }
{?} (myhash..Insert)$(X.1 2);          { try to do it one more time }
{?} (myhash..Insert)$(X.10 );          { try to insert another value
                                         for the same key }
{?} (myhash..insert)$(Z.1);            { use the built-in insert method }
{?} (myhash..insert)$(Z.1);            { insert same value again }
{?} (myhash..insert)$(Z.2);            { also insert a different value }
{?} (myhash..find)$X;                  { show all key-value pairs of X
```

```
                                        (only 1) }
{?} (myhash..find)$Z;                   { show all key-value pairs of Z.(3) }
{?} (myhash..remove)$Z:?values;
{?} :?myhash;                           { get rid of the hash table }
```

`(myhash..ISO)$`

Make all key access case-insensitive. This applies to the full Unicode table, but defaults to ASCII and the upper 128 characters in the ISO 8859-1 (Latin 1) character set if the characters are not UTF-8 encoded.

`(myhash..casesensitive)$`

Make all key access case sensitive. This the default.

`(myhash..forall)$Function`

Apply the function to all key-value pairs. The function can be specified by its name or by its function body. The `forall` method finishes when all elements are traversed or before that if the function fails. The behaviour of `forall` is undefined if the hash table is changed or deleted during the traversal, although this can be done safely. For example, some members may be missed and others may be processed more than once.

Example:

```
{?} new$hash:?myhash;           { create a hash table myhash }
{?} (myhash..insert)$(X.12);    { insert the value 12 for key X }
{?} (myhash..insert)$(Z.1);     { use the built-in insert method }
{?} ( (myhash..forall)          { output all key-value pairs }
    $ (
      =   Key,Value,loop
        .   ( loop
            =   !arg:(?Key.?Value) ?arg
              & out$(str$("Key=" !Key  " Value=" !Value))
              & !loop
            )
          & ~!loop
      )
    )
```

15 Character set

Bracmat supports UTF-8 as well as ISO 8859-1 encoded source code. In most cases it does not matter how characters are encoded, because Bracmat merely sees sequences of

non-zero bytes. Only a few functions explicitly handle UTF-8 encoded characters. When lower- or uppercasing text, Bracmat assumes that the argument is UTF-8 encoded, but graciously falls back to regarding the argument as ISO 8859-1 encoded if parsing the argument as UTF-8 fails. UTF-16 and UTF-32 is not supported, because those encodings make use of zero-bytes.

16 Predefined variables

There are three variables that are predefined, but that in no way are preserved by Bracmat.

```
{?} !w
```

This returns article 11 of the GNU General Public License, Version 2.

```
{?} !c
```

This returns article 12 of the GNU General Public License, Version 2.¡/p¿

```
{?} !v
```

This returns the current version number and build date of Bracmat.

Index