

# An informal tutorial on Joy<sup>1</sup>

Manfred von Thun

---

**Abstract:** Joy is a functional programming language which is not based on the application of functions to arguments but on the composition of functions. It does not use lambda-abstraction of expressions but instead it uses quotation of expressions. A large number of combinators are used to perform dequotation, they have the effect of higher order functions. Several of them can be used to eliminate recursive definitions. Programs in Joy are compact and often look just like postfix notation. Writing programs and reasoning about them is made easy because there is no substitution of actual for formal parameters. This tutorial gives a pragmatic introduction without reference to the theory of Joy.

**Keywords:** functional programming, higher order functions, composition of functions, combinators, elimination of recursive definitions, variable free notation

---

## Introduction

Although the theory of Joy is of interest, this tutorial exposition avoids theory as much as possible.

The remainder of this paper is organised as follows: The next two sections introduce the basic data types and operations on them. One section deals with the simple data types: integers, characters and truth values. The other section deals with aggregate data types: sets, strings and lists. The section after that returns to the central feature of Joy: quotations of programs and their use with combinators. After a short section on definitions the next section resumes the discussion of combinators, in particular those that can eliminate the need for recursive definitions. In the final section several short programs and one larger program are used to illustrate programming with aggregates in Joy.

To add two integers, say 2 and 3, and to write their sum, you type the program

```
2 3 +
```

This is how it works internally: the first numeral causes the integer 2 to be pushed onto a stack. The second numeral causes the integer 3 to be pushed on top of that. Then the addition operator pops the two integers off the stack and pushes their sum, 5. The system reads inputs like the above and executes them when they are terminated by a period ".", like this:

```
2 3 + .
```

In the default mode there is no need for an explicit output instruction, so the numeral 5 is now written to the output file which normally is the screen. So, in the default

---

<sup>1</sup> <http://www.latrobe.edu.au/philosophy/phimvt/joy.html>

mode the terminating "." may be taken to be an instruction to write the top element of the stack. In what follows the terminating period will not be shown any further.

To compute the square of an integer, it has to be multiplied by itself. To compute the square of the sum of two integers, the sum has to be multiplied by itself. Preferably this should be done without computing the sum twice. The following is a program to compute the square of the sum of 2 and 3:

```
2 3 + dup *
```

After the sum of 2 and 3 has been computed, the stack just contains the integer 5. The `dup` operator then pushes another copy of the 5 onto the stack. Then the multiplication operator replaces the two integers by their product, which is the square of 5. The square is then written out as 25. Apart from the `dup` operator there are several others for re-arranging the top of the stack. The `pop` operator removes the top element, and the `swap` operator interchanges the top two elements.

A *list* of integers is written inside square brackets. Just as integers can be added and otherwise manipulated, so lists can be manipulated in various ways. The following concatenates two lists:

```
[1 2 3] [4 5 6 7] concat
```

The two lists are first pushed onto the stack. Then the `concat` operator pops them off the stack and pushes the list [1 2 3 4 5 6 7] onto the stack. There it may be further manipulated or it may be written to the output file.

Joy makes extensive use of *combinators*. These are like operators in that they expect something specific on top of the stack. But unlike operators they execute what they find on top of the stack, and this has to be the *quotation* of a program, enclosed in square brackets. One of these is a combinator for `mapping` elements of one list via a function to another list. Consider the program

```
[1 2 3 4] [dup *] map
```

It first pushes the list of integers and then the quoted program onto the stack. The `map` combinator then removes the list and the quotation and constructs another list by applying the program to each member of the given list. The result is the list [1 4 9 16] which is left on top of the stack.

In *definitions* of new functions no formal parameters are used, and hence there is no substitution of actual parameters for formal parameters. After the following definition

```
square == dup *
```

the symbol `square` can be used in place of `dup *`.

As in other programming languages, definitions may be recursive, for example in the definition of the factorial function. That definition uses a certain recursive pattern that is useful elsewhere. In Joy there is a combinator for *primitive recursion* which has this pattern built in and thus avoids the need for a definition. The `primrec` combinator expects two quoted programs in addition to a data parameter. For an integer data parameter it works like this: If the data parameter is zero, then the first quotation has to produce the value to be returned. If the data parameter is positive then the second has to combine the data parameter with the result of applying the function to its predecessor. For the factorial function the required quoted programs are very simple:

```
[1] [*] primrec
```

computes the factorial recursively. There is no need for any definition. For example, the following program computes the factorial of 5:

```
5 [1] [*] primrec
```

It first pushes the number 5 and then it pushes the two short quoted programs. At this point the stack contains three elements. Then the `primrec` combinator is executed. It pops the two quotations off the stack and saves them elsewhere. Then `primrec` tests whether the top element on the stack (initially the 5) is equal to zero. If it is, it pops it off and executes one of the quotations, the `[1]` which leaves 1 on the stack as the result. Otherwise it pushes a decremented copy of the top element and recurses. On the way back from the recursion it uses the other quotation, `[*]`, to multiply what is now a factorial on top of the stack by the second element on the stack. When all is done, the stack contains 120, the factorial of 5.

As may be seen from this program, the usual branching of recursive definitions is built into the combinator. The `primrec` combinator can be used with many other quotation parameters to compute quite different functions. It can also be used with data types other than integers.

Joy has many more combinators which can be used to calculate many functions without forcing the user to give recursive or non-recursive definitions. Some of the combinators are more data-specific than `primrec`, and others are far more general.

## Integers, characters and truth values

The data types of Joy are divided into simple and aggregate types. The *simple* types comprise integers, characters and the truth values. The aggregate types comprise sets, strings and lists. Literals of any type cause a value of that type to be pushed onto the stack. There they can be manipulated by the general stack operations such as `dup`, `pop` and `swap` and a few others, or they can be manipulated by operators specific to their type. This section introduces literals and operators of the simple types.

An *integer* is just a whole number. Literals of this type are written in decimal notation. The following binary operations are provided:

```
+      -      *      /      %
```

The first four have their conventional meaning, the last is modulus. Operators are written after their operands. Binary operators remove two values from the top of the stack and replace them by the result. For example, the program

```
20 3 4 + * 6 - 100 %
```

evaluates to 34, and this value is left on top of the stack. There are also some unary operators specific to integers like the `abs` operator which takes the absolute value, and the `signum` operator which yields -1, 0 or +1, depending on whether its parameter is negative, zero or positive.

A *character* is a letter, a digit, a punctuation character, in fact any printable character or one of a few white space characters. Literals of type character are written as a single quote followed by the character itself. Values of type character are treated very much like small numbers. That means that other numbers can be added to them, for example 32 to change letter from upper case to lower case. There are two unary

operators which are defined on characters and on integers: `pred` takes the predecessor, `succ` takes the successor. For example,

```
'A 32 + succ succ
```

evaluates to 'c, the third lower case letter.

The type of *truth values* is what in some languages is called *Boolean*. The following are the two literals, the unary negation operator and two binary operators for conjunction and disjunction:

```
true      false      not      and      or
```

For example, the program

```
false true false not and not or
```

evaluates to false.

Values of type integer and character can be compared using the following *relational operators*:

```
=      <      >      #      <=     >=
```

As all other operators, they are written in postfix notation. The result is always a truth value. For example,

```
'A 'E < 2 3 + 15 3 / = and
```

evaluates to true.

## Sets, strings and lists

The *aggregate* types are the unordered type of sets and the ordered types of strings and lists. Aggregates can be built up, combined, taken apart and tested for membership. This section introduces literals and operators of the aggregate types.

A *set* is an unordered collection of zero or more small integers. Literals of type set are written inside curly braces, and the empty set is written as an empty pair of braces. For set literals the ordering of the elements is irrelevant and duplication has no effect. The operators for conjunction and disjunction are also defined on sets. For example, the two equivalent programs

```
{1 3 5 7} {2 4 6 8} or {} or {3 4 5 6 7 8 9 10} and  
{3 7 5 1} {2 4 6 8} or {} or {3 4 5 6 7 8 9 10 10} and
```

evaluate to {3 4 5 6 7 8}. The negation operator `not` takes complements relative to the largest expressible set, which in most implementations will have a maximum of 32 members: from 0 to 31.

A *string* is an ordered sequence of zero or more characters. Literals of this type string are written inside double quotes, and the empty string is written as two adjacent double quotes with nothing inside: `" "`. Note that this is different from the string containing just the blank: `" "`. Two strings can be concatenated, and a string can be reversed. For example,

```
"dooG" reverse " morning" " " concat concat "world" concat
```

evaluates to "Good morning world".

A *list* is an ordered sequence of zero or more values of any type. Literals of type list are written inside square brackets, the empty list is written as an empty pair of brackets. Lists can contain lists as members, so the type of lists is a recursive data type.

Values of the aggregate types, namely sets, strings and lists can be constructed from existing ones by adding a new member with the `cons` operator. This is a binary operator for which the first parameter must be a possible new member and the second parameter must be an aggregate. For sets the new member is added if it is not already there, and for strings and lists the new member is added in front. Here are some examples. The programs on the left evaluate to the literals on the right.

```

5 3 {2 1} cons cons 3 swap cons           {1 2 3 5}
'E 'C "AB" cons cons 'C swap cons         "CECAB"
5 [6] [1 2] cons cons 'A swap cons        ['A 5 [6] 1 2]

```

As the examples show, the `cons` operator is most useful for adding elements into an aggregate which are already on the stack below the aggregate. To add new elements that have just been pushed, the new elements and the aggregate have to be swapped first before the new element can be consed into the aggregate. To facilitate this, Joy has another operator, `swons`, which first performs a `swap` and then a `cons`.

Whereas the `cons` and `swons` operators builds up aggregate values, the two unary operators `first` and `rest` take them apart. Both are defined only on non-empty aggregate values. For the two ordered aggregate types, strings and lists, the meaning is obvious: the `first` operator returns the first element and the `rest` operator returns the string or list without the first element:

```

"CECAB" first                               'C
"CECEB" rest                                "ECAB"
['A 5 [6] 1 2] first                         'A
['A 5 [6] 1 2] rest                          [5 [6] 1 2]

```

But sets are unordered, so it does not make sense to speak of their first members *as sets*. But since their members are integers, the ordering on the integers can be used to determine what the first member is. Analogous considerations apply to the `rest` operator.

```

{5 2 3} first                               2
{5 2 3} rest                                {3 5}

```

For all three types of aggregates the members other than the first can be extracted by repeatedly taking the `rest` and finally the `first` of that. This can be cumbersome for extracting member deep inside. An alternative is to use the `at` operator to *index* into the aggregate, by extracting a member **at** a numerically specified position. For example, the following are two equivalent programs to extract the fifth member of any aggregate:

```

rest rest rest rest first
5 at

```

There is a unary operator which determines the `size` of any aggregate value. For sets this is the number of members, for strings its is the length, and for lists it is the length counting only top level members. The `size` operator yields zero for empty aggregates and a positive integer for others. There is also a unary `null` operator, a predicate

which yields the truth value `true` for empty aggregates and `false` for others. Another predicate, the `small` operator, yields `true` just in case the size is 0 or 1.

Apart from the operators which only affect the stack, there are two for explicit input and output. The `get` operator reads an item from the input file and pushes it onto the stack. The `put` operator pops an item off the stack and writes it to the screen or whatever the output file is. The next program reads two pairs of integers and then compares the sum of the first pair with the sum of the second pair.

```
get get + get get + > put
```

The two `get` operators attempt to read two items and push them onto the stack. There they are immediately added, so they have to be integers. This is repeated for the second pair. At this point the stack contains the two sums. Then the comparison operator pops the two integers and replaces them by a truth value, `true` or `false`, depending on whether the first sum is less than the second sum. The `put` operator pops that truth value and writes it. The stack is now as it was before the program was run, only the input file and the output file are changed.

For another example, the following conducts a silly little dialogue:

```
"What is your name?" put "Hello, " get concat put
```

First the question string is pushed on the stack and then popped to be written out to the screen. Then the `"Hello, "` string is pushed. Next, the `get` operator reads an item from the keyboard and pushes it onto the stack. That item has to be another string, because it will be concatenated with what is below it on the stack. The resultant string is then written out. So, if in answer to the question a user types `"Pat"`, the program finally writes out `"Hello, Pat"`.

## Quotations and Combinators

Lists are really just a special case of *quoted programs*. Lists only contain values of the various types, but quoted programs may contain other elements such as operators and some others that are explained below. A *quotation* can be treated as passive data structure just like a list. For example,

```
[ + 20 * 10 4 - ]
```

has size 6, its second and third elements are 20 and \*, it can be reversed or it can be concatenated with other quotations. But passive quotations can also be made active by *dequotation*.

If the above quotation occurs in a program, then it results in the quotation being pushed onto the stack - just as a list would be pushed. There are many other ways in which that quotation could end up on top of the stack, by being concatenated from its parts, by extraction from a larger quotation, or by being read from the input. No matter how it got to be on top of the stack, it can now be treated in two ways: passively as a data structure, or actively as a program. The square brackets prevented it from being treated actively. Without them the program would have been executed: it would expect two integers which it would add, then multiply the result by 20, and finally push 6, the difference between 10 and 4.

Joy has certain devices called *combinators* which cause the execution of quoted programs that are on top of the stack. This section describes only a very small proportion of them.

One of the simplest is the `i` combinator. Its effect is to execute a single program on top of the stack, and nothing else. Syntactically speaking, its effect is to remove the quoting square brackets and thus to expose the quoted program for execution. Consequently the following two programs are equivalent:

```
[ + 20 * 10 4 - ] i
+ 20 * 10 4 -
```

The `i` combinator is mainly of theoretical significance, but it is used occasionally. The many other combinators are essential for programming in Joy.

One of the most well-known combinators is for branching. The `ifte` combinator expects three quoted programs on the stack, an if-part, a then-part and an else-part, in that order, with the else-part on top. The `ifte` combinator removes and saves the three quotations and then performs the following on the remainder of the stack: It executes the if-part which should leave a truth value on top of the stack. That truth value is saved and the stack is restored to what it was before the execution of the if-part. Then, if the saved truth value was `true`, the `ifte` combinator executes the then-part, otherwise it executes the else-part.

In most cases the three parts would have been pushed in that order just before the `ifte` combinator is executed. But any or all of the three parts could have been constructed from other quotations.

In the following example the three parts are pushed just before the `ifte` combinator is executed. The program looks at a number on top of the stack, and if it is greater than 1000 it will halve it, otherwise it will triple it.

```
[1000 >] [2 /] [3 *] ifte
```

Some combinators require that the stack contains values of certain types. Many are analogues of higher order functions familiar from other programming languages: `map`, `filter` and `fold`. Others only make sense in Joy. For example, the `step` combinator can be used to access all elements of an aggregate in sequence. For strings and lists this means the order of their occurrence, for sets it means the underlying order. The following will `step` through the members of the second list and `swons` them into the initially empty first list. The effect is to reverse the non-empty list, yielding `[5 6 3 8 2]`.

```
[] [2 8 3 6 5] [swons] step
```

The `map` combinator expects an aggregate value on top of the stack, and it yields another aggregate of the same size. The elements of the new aggregate are computed by applying the quoted program to each element of the original aggregate. An example was already given in the introduction.

Another combinator that expects an aggregate is the `filter` combinator. The quoted program has to yield a truth value. The result is a new aggregate of the same type containing those elements of the original for which the quoted program yields `true`. For example, the quoted program `['z >]` will yield truth for characters whose

numeric values is greater than that of `z`. Hence it can be used to remove upper case letters and blanks from a string. So the following evaluates to "ohnmith":

```
"John Smith" ['Z >] filter
```

Sometimes it is necessary to add or multiply or otherwise combine all elements of an aggregate value. The `fold` combinator can do just that. It requires three parameters: the aggregate to be folded, the quoted value to be returned when the aggregate is empty, and the quoted binary operation to be used to combine the elements. In some languages the combinator is called `reduce` (because it turns the aggregate into a single value), or `insert` (because it looks as though the binary operation has been inserted between any two members). The following two programs compute the sum of the members of a list and the sum of the squares of the members of a list. They evaluate to 10 and 38, respectively.

```
[2 5 3] 0 [+] fold
[2 5 3] 0 [dup * +] fold
```

To compute the average or arithmetic mean of the members of a set or a list, we have to divide the sum by the size. (Because of the integer arithmetic, the division will produce an inaccurate average.) The aggregate needs to be looked at twice: once for the sum and once for the size. So one way to compute the average is to duplicate the aggregate value first with the `dup` operator. Then take the `sum` of the top version. Then use the `swap` operator to interchange the position of the sum and the original aggregate, so that the original is now on top of the stack. Take the size of that. Now the stack contains the sum and the size, with the size on top. Apply the division operator to obtain the average value.

```
dup 0 [+] fold swap size /
```

One nice feature of this little program is that it works equally for set values as for list values. This is because the constituents `fold` and `size` work for both types.

But there are two aspects of this program which are unsatisfactory. One concerns the `dup` and `swap` operators which make the program hard to read. The other concerns the sequencing of operations: The program causes the computation of the sum to occur before the computation of the size. But it does not matter in which order they are computed, in fact on a machine with several processors the sum and the size could be computed in parallel. Joy has a combinator which addresses this problem: there is one **one** data parameters, the aggregate, which is to be fed to **two** functions. From each of the functions a value is to be **constructed**. Because of the one datum and the two functions, the combinator is called `constr12`. The program for the average looks like this:

```
[0 [+] fold] [size] constr12 /
```

There are several similar combinators with different combinations of trailing digits.

## Definitions

In conventional languages the definition of a function of one or more arguments has to name these as formal parameters `x`, `y` ... For example, the squaring function might be defined by some variation of any of the following:

```

square(x) = x * x
(defun (square x) (* x x))
square = lambda x.x * x

```

In Joy formal parameters such as `x` above are not required, a definition of the squaring function is simply

```
square == dup *
```

This is one of the principal differences between Joy and those languages that are based on the lambda calculus. The latter include (the purely functional subsets of) Lisp, Scheme, ML, Miranda<sup>2</sup> and Haskell. All of these are based on the application of functions to arguments or actual parameters.

In definitions and abstractions of functions the formal parameters have to be named - `x`, `y` and so on, or something more informative. This is different in Joy. It is based on the composition of functions and not on the application of functions to arguments. In definitions and abstractions of functions the arguments do not need be named and as formal parameters indeed cannot be named. One consequence is that there are no *environments* of name-value pairs. Instead the work of environments is done by higher order functions called combinators.

Finally, the concrete syntax of the language is an integral part of the language and aids in reasoning about Joy programs in the metalanguage.

Suppose it is required to transform a list of numbers into the list of their cubes. The cube of a single number is of course computed by

```
dup dup * *
```

It would be possible to introduce a definition of the cube function. But that would introduce another name, `cube`. If the cube function is used only once for computing the cubes of a list of numbers, then it may not be desirable to give a definition of it at all. In Joy the list of cubes is computed by the first line below, but it is also possible to give an explicit definition as in the second line.

```
[dup dup * *] map
cubelist == [dup dup * *] map
```

In a language that is based on the lambda calculus both would need a lambda abstraction with a variable, say `x`, for the number to be cubed. And of course the second line would need an additional formal parameter, say `l`, or a lambda abstraction with a variable `l` for the list to which the cubelist function is to be applied.

Suppose now that it is required to transform a *list of lists* of numbers into the *list of lists* of their cubes. One might give the definition

```
cubelistlist == [ [dup dup * *] map ] map
```

Of course, if that function is only to be used once, one might not bother to give a definition at all but use the right hand side directly. In languages based on abstraction, at least two formal parameters are needed just for the right hand side, and another for the definition itself. For example, in Scheme the definition looks like this:

---

<sup>2</sup> "Miranda" is a trademark of Research Software Ltd.

```

(define (cubelistlist ll)
  (map (lambda (l)
        (map (lambda (n) (* n (* n n)))
              l ) )
        ll )

```

Here the two formal parameters are `n` for the number and `l` for the list of numbers on the right hand side, and `ll` for the list of lists of numbers in the definition itself.

As in other languages, definitions can be recursive in Joy. In the first line below is a recursive definition of the factorial function in one of many variants of conventional notation. In the second line is a recursive definition in Joy.

```

factorial(x) = if x = 0 then 1 else x * factorial(x - 1)
factorial == [0 =] [pop 1] [dup 1 - factorial *] ifte

```

Again the Joy version does not use a formal parameter `x`. It works like this: The definition uses the `ifte` combinator immediately after the if-part, the then-part and the else-part have been pushed.

The `ifte` combinator then does this: it executes the if-part, in this case `[0 =]`, which tests whether the (anonymous) integer parameter is equal to zero. If it is, then the else-part is executed, in this case `[pop 1]`, which pops the parameter off the stack and replaces it by one. Otherwise the else-part is executed, in this case `[dup 1 - factorial *]`. This uses `dup` to make another copy of the parameter and subtracts one from the copy. Then the `factorial` function is called recursively on that. Finally the original parameter and the just computed factorial are multiplied.

The definition could be shortened and made a little more efficient by using the inbuilt predicate `null` which tests for zero and the `pred` operator which takes the predecessor of a number. But these changes are insignificant.

For more complex functions of several arguments it is necessary to be able to access the arguments anywhere in the definition. Joy avoids formal parameters altogether, and hence in general arbitrary access has to be done by mechanisms more sophisticated than `dup`, `swap` and `pop`.

Here are some more definitions that one might have:

```

sum == 0 [+] fold
product == 1 [*] fold
average == [sum] [size] constr12 /
concatenation == "" [concat] fold

```

The last definition is for an operator which yields a single string which is the concatenation of a list of strings.

## Recursive Combinators

If one wanted to compute the list of factorials of a given list, this could be done by

```
[ factorial ] map
```

But this relies on an external definition of `factorial`. It was necessary to give that definition explicitly because it is recursive. If one only wanted to compute factorials of lists of numbers, then it would be a minor nuisance to be forced to define `factorial` explicitly just because the definition is recursive.

A high proportion of recursively defined functions exhibit a very simple pattern: There is some test, the if-part, which determines whether the ground case obtains. If it does, then the non-recursive then-part is executed. Otherwise the recursive else-part has to be executed. In the else-part there is only *one* recursive call, and there can be something before the recursive call and something after the recursive call. It helps to think of the else-part to have two components, the else1-part before the recursive call, and the else2-part after the recursive call. This pattern is called *linear recursion*, and it occurs very frequently.

Joy has a useful device, the `linrec` combinator, which allows computation of anonymous functions that *might* have been defined recursively using a linear recursive pattern. Whereas the `ifte` combinator requires three quoted parameters, the `linrec` combinator requires four: an if-part, a then-part, an else1-part and an else2-part. For example, the factorial function could be computed by

```
[null] [succ] [dup pred] [*] linrec
```

There is no need for a definition, the above program can be used directly.

Very frequently the if-part of a linear recursion tests for a simple base condition which depends on the type of the parameter. For numbers that condition tends to be being zero, for sets, strings and lists that condition tends to be being empty. The else1-part frequently makes the parameter smaller in some way. For numbers it decrements them, for sets, strings and lists it takes the `rest`.

Joy has another useful combinator which has the appropriate if-part and else1-part built in. This is the `primrec` combinator, which only has to be supplied with two quotation parameters, the (modified) then-part and the else2-part of linear recursion. For the factorial function the two quotation parameters are very simple:

```
[1] [*] primrec
```

computes the factorial function. So, if one wanted to compute the list of factorial of a given list of numbers this can be done by either of the following:

```
[ [null] [succ] [dup pred] [*] linrec ] map
[ [1] [*] primrec ] map
```

The factorial of a number is the product of successive natural numbers up to the actual parameter. The following compute instead their sums and the sum of their squares:

```
[0] [+] primrec
[0] [dup * +] primrec
```

Many of the Joy combinators are polymorphic in the sense that they can be applied to parameters of quite different types. The combinator `primrec` can be applied not only to numbers but also to lists. For example, applied to the list `[1 2 3]` the program

```
[[]] [[] cons cons] primrec
```

produces the list `[1 [2 [3 []]]]`. Lisp programmers will recognise a similarity to "dotted pairs". In the following, the first turns a set of numbers into a list, the second turns a list of numbers into a set:

```
[[]] [cons] primrec
[{}] [cons] primrec
```

In fact, the first can also be applied to a list and the second can also be applied to a set. But in that case they just compute the identity. They can even be applied to

numbers - and then they produce a list or a set of numbers from the parameter down to 1.

In many recursive definitions there are two recursive calls of the function being defined. This is the pattern of *binary recursion*, and it is used in the usual definitions of quicksort and of the Fibonacci function. Joy has a facility that eliminates the need for a recursive definition, the `binrec` combinator.

The following will *quicksort* a list whose members can be a mixture of anything except lists. The program easily fits onto one line, but for reference it is here written over several numbered lines:

```
1      [small]
2      []
3      [uncons [>] split]
4      [[swap] dip cons concat]
5      binrec
```

This is how it works: Lines 1..4 each push a quoted program. In line 5 the `binrec` combinator is called, and it will make use of the four quoted programs and below that the list to be sorted. The four quoted programs are saved elsewhere, and the `binrec` combinator begins by executing the program from line 1. This tests whether the list to be sorted is small, i.e. has at most one member. If indeed it is small, then it is sorted already.

The `binrec` combinator now executes the program from line 2, which does nothing and hence leaves the small list as it is. On the other hand, if the list is not small, then the programs in lines 3 and 4 will be executed. The program in line 3 removes the first element from the list and uses it as a pivot to split the rest of the list into two sublists, by using the comparison function in `[>]` and the `split` combinator.

At this point the `binrec` combinator calls itself recursively on the two sublists and sorts them both. Finally the program in line 4 combines the two sorted versions and the original pivot into a single sorted list. The three items are not quite in the required order, so the `[swap] dip` part puts the pivot in between the two sorted lists.

Then `cons` puts the pivot in front of the topmost string or list, and finally `concat` combines everything into one single sorted list. Since all operations in the program also work on strings, the program itself can equally well be used to sort a string.

In fact, the program can be used on sets too, but this of course is pointless. The program is useful, it is part of the Joy system library under the name of `qsort`.

Many other functions are often defined by recursive definitions that use binary recursion. In Joy they can all be computed with the `binrec` combinator without the need for a definition. For example, the following computes the *Fibonacci* function; it implements the usual inefficient algorithm:

```
[small] [] [pred dup pred] [+] binrec
```

The system library of course contains the well known efficient algorithm.

There are only a few second order combinators, ones which require a first order combinator as parameter. One of the is `treerec` for recursing through *trees*. These are either anything but a list, or lists of trees. For example, in the following `treerec` is given `[map]` as a parameter, which in turn will be given `[dup *]` as a parameter when

`treerec` encounters a list. The function to be applied to numbers possibly deeply embedded within lists is the squaring function `[dup *]`.

Here is an example:

```
[ 1 [2 3] [[[4]]] 5 ] [dup *] [map] treerec
```

produces

```
[ 1 [2 9] [[[16]]] 25 ]
```

All of these combinators can be defined in other functional languages, but they are less useful there. This is because their parameters have to be abstractions with variables, and not quotations as in Joy.

## Programming with aggregates

The *aggregate* types of Joy are lists, sets and strings. There are several unary operators which take an aggregate as parameter and produce as value a list of subaggregates. One of these is the `powerlist` operator. For an aggregate of size  $N$  it produces a list of all the  $2^N$  subaggregates.

Here is an example:

```
[1 2 3] powerlist
```

produces as result

```
[ [1 2 3] [1 2] [1 3] [1] [2 3] [2] [3] [] ]
```

If the ordering does not suit, the result list can always be rearranged, for example it can be reversed. For another example, one can sort the list according to size. The `mk_qsort` combinator expects an aggregate and a quoted operator as parameters and it applies the operator to each member of the aggregate to use as the basis for sorting them.

```
[1 2 3] powerlist [size] mk_qsort
```

produces as a result

```
[ [] [1] [2] [3] [1 2] [1 3] [2 3] [1 2 3] ]
```

The `powerlist` operators can also be applied to a string. The result is a list of all substrings. In the following the result list is *filtered* to retain only those substrings whose size is greater than 3. This is achieved by the `filter` combinator which expects an aggregate and a quoted predicate. The first line is the program, the second line is the result:

```
"abcde" powerlist [size 3 >] filter  
[ "abcde" "abcd" "abce" "abde" "acde" "bcde" ]
```

The `powerlist` operators can also be applied to a set. In the program on the first line below the list of subsets is then filtered to retain only those of size 3; the result is the list of subsets in the second line:

```
{1 2 3 4} powerlist [size 3 =] filter  
[ {1 2 3} {1 2 4} {1 3 4} {2 3 4} ]
```

Suppose it is required to find the list, in ascending order, of all sums of any three distinct numbers taken from a given set of numbers. We already know how to get the list of all three-membered subsets. Each should be replaced by its sum, and that can be done with the `map` combinator applied to the whole list. The resulting list of sums then needs to be sorted. The example in the first line does just that, giving the result in the second line:

```
{1 2 3 4 5} powerlist [size 3 =] filter [sum] map qsort
[6 7 8 8 9 9 10 10 11 12]
```

In the remainder of this section a small program is to be constructed which takes one sequence as parameter and returns the list of all *permutations* of that sequence. Here is a first draft:

```
1   If S has only zero or one member
2   then it has only one permutation, so take its unit list
3   else take the first and rest of S,
        recurse to construct the permutations of the rest
4   insert the first in all positions in all permutations
```

The recursion pattern is linear, so we can use the `linrec` combinator to arrive at this first incomplete program:

```
1   [ small ]
2   [ unitlist ]
3   [ uncons ]
4   [ "insert the first in all positions in all permutations" ]
5   linrec
```

The anonymous recursion between steps 3 and 4 will have left a list of permutations of the rest of `s` on top of the stack.

Next, it is necessary to insert the original first of `s` into all positions into all these resulting permutations. This involves replacing each single permutation by a list of permutations with the original first inserted in all places.

This calls for the `map` combinator to apply a *constructed program* to each permutation. The original first is currently the second item on the stack. to make it available to the program to be constructed, it is `swapped` to the top. The required program consists of a constant part and a variable part.

The constant part now has to be pushed onto the stack. Then the first is `consed` into the required program. Then `map` will create a list of list of permutations. But this is a two-level list, and it should be one-level. So the two level list has to be flattened to a one-level list.

```
4.1 [ swap
4.2   [ "the constant part of the program" ]
4.3   cons map
4.4   "flatten the resulting list of lists of sequences" ]
```

The constant part of the constructed program has to be written next. The constructed program will be used to `map` all permutations of the rest, and in each case it will begin by pushing the original first on top of the current permutation being mapped. It then has to insert this first into all positions of the current permutation.

This again calls for a linear recursion with `linrec`. One way to do this is to give this anonymous recursive function just one parameter, the current permutation with the original first `swons` in as an initial element. So the task is now to insert this initial element into all positions in the remainder which is the current permutation.

```

4.2.2.1  If the current sequence is small
4.2.2.2      then return just its unit list
4.2.2.3      else keep 1. a copy
                    2. its second and
                    3. the sequence without its second
                    anonymously recurse on 3.
4.2.2.4      construct a program to insert the second
                    use map to do the insertion
                    use cons to add the copy from 1.

```

So the constant part 4.2 looks like this:

```

4.2.1      [ swons
4.2.2.1    [ small ]
4.2.2.2    [ unitlist ]
4.2.2.3    [ dup unswons [uncons] dip swons ]
4.2.2.4    [ swap [swons] cons map cons ]
4.2.2.5    linrec ]

```

The only other part that needs to be written is for flattening. This should be trivial by now: If the list is small, then take its unit list else take its first and its rest anonymously recurse on the rest, concatenate the saved first into the result.

Here is the required program:

```

4.4      [ null ] [ ] [ uncons ] [ concat] linrec

```

The entire program now is the following:

```

1      [ small ]
2      [ unitlist ]
3      [ uncons ]
4.1    [ swap
4.2.1  [ swons
4.2.2.1 [ small ]
4.2.2.2 [ unitlist ]
4.2.2.3 [ dup unswons [uncons] dip swons ]
4.2.2.4 [ swap [swons] cons map cons ]
4.2.2.5 linrec ]
4.3    cons map
4.4    [null] [] [uncons] [concat] linrec ]
5      linrec.

```

An essentially identical program is in the Joy library under the name `permlist`. It is considerably shorter than the one given here because it uses two subsidiary programs `insertlist` and `flatten` which are useful elsewhere. The program given above is an example of a non-trivial program which uses the `linrec` combinator three times and the `map` combinator twice, with *constructed programs* as parameters on both occasions.

Of course such a program can be written in lambda calculus languages such as *Lisp*, *Scheme*, *ML* or *Miranda*, but it would need many recursive definitions with attendant named formal parameters.