KASH

GAP3 Programming Language

KANT–Group
Technische Universität Berlin
Institut für Mathematik, MA 8-1
Straß e des 17. Juni 136
10623 Berlin, Germany

http://www.math.tu-berlin.de/~kant

October 18, 2005

Contents

Chapter 1

Preface

This description of the GAP3 programming language was taken from the GAP3 manual by Martin Schönert together with Hans Ulrich Besche, Thomas Breuer, Frank Celler, Bettina Eick, Volkmar Felsch, Alexander Hulpke, Jürgen Mnich, Werner Nickel, Alice Niemeyer, Götz Pfeiffer, Udo Polis, and Heiko Theißen.

While KASH3 uses the GAP3 shell as its user interface not all examples given here will work in KASH3. Nevertheless this is a more detailed description of the original features of the GAP3 programming language than provided in the *Introduction to KASH3*. Please refer to that document for the new features like methods, maps, and extended types.

KANT V4 is a program library for computations in algebraic number fields and (global) algebraic function fields. In the number field case, algebraic integers are considered to be elements of a specified order of an appropriate field $\mathcal{F}$. Algebraic numbers are presented by an integer and a denominator, usually chosen as a natural number. In the function field case, also orders of F are used, but now over different coefficient rings. The representation of algebraic functions is then done in an analogous way as for algebraic numbers. The available algorithms provide the user with the means to compute many invariants of $\mathcal{F}$. In the number field case it is possible to solve tasks like calculating the solutions of Diophantine equations related to $\mathcal{F}$. Further subfields of $\mathcal{F}$ can be generated and $\mathcal{F}$ can be embedded into an overfield. The potential of moving elements between different fields (orders) is a significant feature of our system. In the function field case, for example, genus computations and the construction of Riemann-Roch spaces are available.

KANT V4 was developed at the University of Düsseldorf from 1987 until 1993 and at the Technical University Berlin afterwards. During these years the performance of existing algorithms and their implementations grew dramatically. While calculations in number fields of degree 4 and up were nearly impossible before 1970 and number fields of degree more than 10 were beyond reach until 1990, it is now possible to compute in number fields of degree well over 20, and – in special cases – even beyond 1000. This also characterizes one of the principles of KANT V4, namely to support computations in number fields of arbitrary degree rather than fixing the degree and pushing the size of the discriminant to the limit.

KANT V4 consists of a C–library of more than 1000 functions for doing arithmetic in number fields. Of course, the necessary auxiliaries from linear algebra over rings, especially lattices, are also included. The set of these functions is based on the computer algebra system MAGMA

from which we adopt our storage management, arithmetic for (long) integers and arbitrary precision floating point numbers, arithmetic for finite fields, polynomial arithmetic and a variety of other tools. Essentially, all of the public domain part of MAGMA is contained in KANT V4. In return, almost all KANT V4 routines are included in MAGMA.

To make KANT V4 easier to use we developed a shell called KASH. This shell is based on that of the group theory package GAP and the handling is similar to that of MAPLE. We put great effort into enabling the user to handle the number theoretical objects in the very same way as one would do using pencil and paper. For example, there is just one command **Factor** for the factorization of elements from a factorial monoid like rational integers in $\mathbb{Z}$, polynomials over a field, or ideals from a Dedekind ring.

The main features of the current release of KASH are

- computation of maximal orders in numbers fields,
- computation of class groups,
- computation of fundamental units in arbitrary orders,
- decomposition of ideals in number fields,
- arithmetic of ideals,
- arithmetic of relative extensions of number fields,
- computation of maximal orders of a relative extension
- computation of normal forms of modules in relative extensions,
- solution of norm equations in absolute and relative extensions,
- computation of subfields,
- computation of Galois groups up to degree 15,
- computation of automorphisms in normal number fields,
- computation of ray class fields,
- computation of the genus of an algebraic function field, handling of places, divisors and Riemann-Roch spaces,
- computation of maximal orders and ideal arithmetic in function fields,
- computation of reduced bases and fundamental units in global function fields,
- arithmetic in relative lattices,
- solving Thue and unit equations, integral points on Mordell curves,
- solving index form equations.

The development of KANT V4 as well as KASH is continued in view of providing the user with the most advanced tools for computations in algebraic number fields. Suggestions for important features to be included are welcome.

1.1 Copyright

TU Berlin
Fachbereich 3 Mathematik
Strasse des 17. Juni 136
10623 Berlin, Germany

Chapter 2

The Programming Language

This chapter describes the GAP programming language. It should allow you in principle to predict the result of each and every input. In order to know what we are talking about, we first have to look more closely at the process of interpretation and the various representations of data involved.

First we have the input to GAP, given as a string of characters. How those characters enter GAP is operating system dependent, e.g., they might be entered at a terminal, pasted with a mouse into a window, or read from a file. The mechanism does not matter. This representation of expressions by characters is called the **external representation** of the expression. Every expression has at least one external representation that can be entered to get exactly this expression.

The input, i.e., the external representation, is transformed in a process called **reading** to an internal representation. At this point the input is analyzed and inputs that are not legal external representations, according to the rules given below, are rejected as errors. Those rules are usually called the **syntax** of a programming language.

The internal representation created by reading is called either an **expression** or a **statement**. Later we will distinguish between those two terms, however now we will use them interchangeably. The exact form of the internal representation does not matter. It could be a string of characters equal to the external representation, in which case the reading would only need to check for errors. It could be a series of machine instructions for the processor on which GAP is running, in which case the reading would more appropriately be called compilation. It is in fact a tree–like structure.

After the input has been read it is again transformed in a process called **evaluation** or **execution**. Later we will distinguish between those two terms too, but for the moment we will use them interchangeably. The name hints at the nature of this process, it replaces an expression with the value of the expression. This works recursively, i.e., to evaluate an expression first the subexpressions are evaluated and then the value of the expression is computed according to rules given below from those values. Those rules are usually called the **semantics** of a programming language.

The result of the evaluation is, not surprisingly, called a **value**. The set of values is of course a much smaller set than the set of expressions; for every value there are several expressions that will evaluate to this value. Again the form in which such a value is represented internally

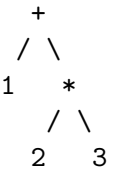
does not matter. It is in fact a tree–like structure again.

The last process is called **printing**. It takes the value produced by the evaluation and creates an external representation, i.e., a string of characters again. What you do with this external representation is up to you. You can look at it, paste it with the mouse into another window, or write it to a file.

Lets look at an example to make this more clear. Suppose you type in the following string of 8 characters

```
1 + 2 * 3;
```

GAP takes this external representation and creates a tree like internal representation, which we can picture as follows



This expression is then evaluated. To do this GAP first evaluates the right subexpression `2*3`. Again to do this GAP first evaluates its subexpressions 2 and 3. However they are so simple that they are their own value, we say that they are self–evaluating. After this has been done, the rule for `*` tells us that the value is the product of the values of the two subexpressions, which in this case is clearly 6. Combining this with the value of the left operand of the `+`, which is self–evaluating too gives us the value of the whole expression 7. This is then printed, i.e., converted into the external representation consisting of the single character `7`.

In this fashion we can predict the result of every input when we know the syntactic rules that govern the process of reading and the semantic rules that tell us for every expression how its value is computed in terms of the values of the subexpressions. The syntactic rules are given in sections 2.1, 2.2, 2.3, 2.4, 2.5, and 2.20, the semantic rules are given in sections 2.6, 2.7, 2.8, 2.9, 2.10, 2.11, 2.12, 2.13, 2.14, 2.15, 2.16, 2.17, 2.18, and the chapters describing the individual data types.

2.1 Lexical Structure

The input of GAP consists of sequences of the following characters.

Digits, uppercase and lowercase letters, *space*, *tab*, *newline*, and the special characters

```
"       '       (       )       *       +       ,       _
.       /       :       ;       <       =       >       ~
[       \       ]       ^       _       {       }       #
```

Other characters will be signalled as illegal. Inside strings and comments the full character set supported by the computer is allowed.

2.2 Symbols

The process of reading, i.e., of assembling the input into expressions, has a subprocess, called **scanning**, that assembles the characters into symbols. A **symbol** is a sequence of characters that form a lexical unit. The set of symbols consists of keywords, identifiers, strings, integers, and operator and delimiter symbols.

A keyword is a reserved word consisting entirely of lowercase letters (see 2.4). An identifier is a sequence of letters and digits that contains at least one letter and is not a keyword (see 2.5). An integer is a sequence of digits. A string is a sequence of arbitrary characters enclosed in double quotes.

Operator and delimiter symbols are

```
+       -       *       /       ^       ~
=       <>      <       <=      >       >=
:=      .       ..      ->      ,       ;
[       ]       {       }       (       )
```

Note that during the process of scanning also all whitespace is removed (see 2.3).

2.3 Whitespaces

The characters *space*, *tab*, *newline*, and *return* are called **whitespace characters**. Whitespace is used as necessary to separate lexical symbols, such as integers, identifiers, or keywords. For example `Thorondor` is a single identifier, while `Th or ondor` is the keyword `or` between the two identifiers `Th` and `ondor`. Whitespace may occur between any two symbols, but not within a symbol. Two or more adjacent whitespaces are equivalent to a single whitespace. Apart from the role as separator of symbols, whitespaces are otherwise insignificant. Whitespaces may also occur inside a string, where they are significant. Whitespaces should also be used freely for improved readability.

A **comment** starts with the character `#`, which is sometimes called sharp or hatch, and continues to the end of the line on which the comment character appears. The whole comment, including `#` and the *newline* character is treated as a single whitespace. Inside a string, the comment character `#` looses its role and is just an ordinary character.

For example, the following statement

```
if i<0 then a:=-i;else a:=i;fi;
```

is equivalent to

```
if i < 0  then      # if i is negative
    a := -i;        #     take its inverse
else                # otherwise
    a := i;         #     take itself
fi;
```

(which by the way shows that it is possible to write superfluous comments). However the first statement is not equivalent to

```
ifi<0thena:=-i;elsea:=i;fi;
```

since the keyword `if` must be separated from the identifier `i` by a whitespace, and similarly `then` and `a`, and `else` and `a` must be separated.

2.4 Keywords

Keywords are reserved words that are used to denote special operations or are part of statements. They must not be used as identifiers. The keywords are

```
and       do        elif      else      end       fi
for       function  if        in        local     mod
not       od        or        repeat    return    then
until     while     quit
```

Note that all keywords are written in lowercase. For example only `else` is a keyword; `Else`, `eLsE`, `ELSE` and so forth are ordinary identifiers. Keywords must not contain whitespace, for example `el if` is not the same as `elif`.

2.5 Identifiers

An identifier is used to refer to a variable (see 2.7). An identifier consists of letters, digits, and underscores _, and must contain at least one letter or underscore. An identifier is terminated by the first character not in this class. Examples of valid identifiers are

```
a                   foo                 aLongIdentifier
hello               Hello               HELLO
x100                100x                _100
some_people_prefer_underscores_to_separate_words
WePreferMixedCaseToSeparateWords
```

Note that case is significant, so the three identifiers in the second line are distinguished.

The backslash `\` can be used to include other characters in identifiers; a backslash followed by a character is equivalent to the character, except that this escape sequence is considered to be an ordinary letter. For example `G\(2\,5\)` is an identifier, not a call to a function `G`.

An identifier that starts with a backslash is never a keyword, so for example `\*` and `\mod` are identifier.

The length of identifiers is not limited, however only the first 1023 characters are significant. The escape sequence *newline* is ignored, making it possible to split long identifiers over multiple lines.

2.6 Expressions

An **expression** is a construct that evaluates to a value. Syntactic constructs that are executed to produce a side effect and return no value are called **statements** (see 2.11). Expressions appear as right hand sides of assignments (see 2.12), as actual arguments in function calls (see 2.8), and in statements.

Note that an expression is not the same as a value. For example `1 + 11` is an expression, whose value is the integer 12. The external representation of this integer is the character sequence `12`, i.e., this sequence is output if the integer is printed. This sequence is another expression whose value is the integer 12. The process of finding the value of an expression is done by the interpreter and is called the **evaluation** of the expression.

Variables, function calls, and integer, permutation, string, function, list, and record literals (see 2.7, 2.8,2.18,), are the simplest cases of expressions.

Expressions, for example the simple expressions mentioned above, can be combined with the operators to form more complex expressions. Of course those expressions can then be combined further with the operators to form even more complex expressions. The **operators** fall into three classes. The **comparisons** are `=`, `<>`, `<=`, `>`, `>=`, and `in` (see 2.9 and 3.13). The **arithmetic operators** are `+`, `-`, `*`, `/`, `mod`, and `^` (see 2.10). The **logical operators** are `not`, `and`, and `or`.

```
gap> 2 * 2;      # a very simple expression with value
4
gap> 2 * 2 + 9 = Fibonacci(7) and  Fibonacci(13) in Primes;
true             # a more complex expression
```

2.7 Variables

A **variable** is a location in a GAP program that points to a value. We say the variable is **bound** to this value. If a variable is evaluated it evaluates to this value.

Initially an ordinary variable is not bound to any value. The variable can be bound to a value by **assigning** this value to the variable (see 2.12). Because of this we sometimes say that a variable that is not bound to any value has no assigned value. Assignment is in fact the only way by which a variable, which is not an argument of a function, can be bound to a value. After a variable has been bound to a value an assignment can also be used to bind the variable to another value.

A special class of variables are **arguments** of functions. They behave similarly to other variables, except they are bound to the value of the actual arguments upon a function call (see 2.8).

Each variable has a name that is also called its **identifier**. This is because in a given scope an identifier identifies a unique variable (see 2.5). A **scope** is a lexical part of a program text. There is the global scope that encloses the entire program text, and there are local scopes that range from the `function` keyword, denoting the beginning of a function definition, to the corresponding `end` keyword. A local scope introduces new variables, whose identifiers are given in the formal argument list and the `local` declaration of the function (see 2.18). Usage of an identifier in a program text refers to the variable in the innermost scope that has this identifier as its name. Because this mapping from identifiers to variables is done when the program is read, not when it is executed, GAP is said to have lexical scoping. The following example shows how one identifier refers to different variables at different points in the program text.

```
g := 0;            # global variable g
```

```
x := function ( a, b, c )
   local   y;
   g := c;         # c refers to argument c of function x
   y := function ( y )
       local  d, e, f;
       d := y;     # y refers to argument y of function y
       e := b;     # b refers to argument b of function x
       f := g;     # g refers to global variable g
       return d + e + f;
   end;
   return y( a );  # y refers to local y of function x
end;
```

It is important to note that the concept of a variable in GAP is quite different from the concept of a variable in programming languages like PASCAL. In those languages a variable denotes a block of memory. The value of the variable is stored in this block. So in those languages two variables can have the same value, but they can never have identical values, because they denote different blocks of memory. (Note that PASCAL has the concept of a reference argument. It seems as if such an argument and the variable used in the actual function call have the same value, since changing the argument's value also changes the value of the variable used in the actual function call. But this is not so; the reference argument is actually a pointer to the variable used in the actual function call, and it is the compiler that inserts enough magic to make the pointer invisible.) In order for this to work the compiler needs enough information to compute the amount of memory needed for each variable in a program, which is readily available in the declarations PASCAL requires for every variable.

In GAP on the other hand each variable justs points to a value.

2.8 Function Calls

function-var`()`
function-var`(` *arg-expr* {`,` *arg-expr*} `)`

The function call has the effect of calling the function *function-var*. The precise semantics are as follows.

First GAP evaluates the *function-var*. Usually *function-var* is a variable, and GAP does nothing more than taking the value of this variable. It is allowed though that *function-var* is a more complex expression, namely it can for example be a selection of a list element *list-var*`[`*int-expr*`]`, or a selection of a record component *record-var*`.`*ident*. In any case GAP tests whether the value is a function. If it is not, GAP signals an error.

Next GAP checks that the number of actual arguments *arg-expr*s agrees with the number of formal arguments as given in the function definition. If they do not agree GAP signals an error. An exception is the case when there is exactly one formal argument with the name `arg`, in which case any number of actual arguments is allowed.

Now GAP allocates for each formal argument and for each formal local a new variable. Remember that a variable is a location in a GAP program that points to a value. Thus for each

formal argument and for each formal local such a location is allocated.

Next the arguments *arg-expr*s are evaluated, and the values are assigned to the newly created variables corresponding to the formal arguments. Of course the first value is assigned to the new variable corresponding to the first formal argument, the second value is assigned to the new variable corresponding to the second formal argument, and so on. However, GAP does not make any guarantee about the order in which the arguments are evaluated. They might be evaluated left to right, right to left, or in any other order, but each argument is evaluated once. An exception again occurs if the function has only one formal argument with the name `arg`. In this case the values of all the actual arguments are stored in a list and this list is assigned to the new variable corresponding to the formal argument `arg`.

The new variables corresponding to the formal locals are initially not bound to any value. So trying to evaluate those variables before something has been assigned to them will signal an error.

Now the body of the function, which is a statement, is executed. If the identifier of one of the formal arguments or formal locals appears in the body of the function it refers to the new variable that was allocated for this formal argument or formal local, and evaluates to the value of this variable.

If during the execution of the body of the function a `return` statement with an expression (see 2.19) is executed, execution of the body is terminated and the value of the function call is the value of the expression of the `return`. If during the execution of the body a `return` statement without an expression is executed, execution of the body is terminated and the function call does not produce a value, in which case we call this call a procedure call (see 2.13). If the execution of the body completes without execution of a `return` statement, the function call again produces no value, and again we talk about a procedure call.

```
gap> Fibonacci( 11 );
    # a call to the function Fibonacci with actual argument 11
89
gap> G.operations.RightCosets( G, Intersection( U, V ) );;
    # a call to the function in G.operations.RightCosets
    # where the second actual argument is another function call
```

2.9 Comparisons

left-expr `=` *right-expr*
left-expr `<>` *right-expr*

The operator `=` tests for equality of its two operands and evaluates to `true` if they are equal and to `false` otherwise. Likewise `<>` tests for inequality of its two operands. Note that any two objects can be compared, i.e., `=` and `<>` will never signal an error. For each type of objects the definition of equality is given in the respective chapter. Objects of different types are never equal, i.e., `=` evaluates in this case to `false`, and `<>` evaluates to `true`.

left-expr `<` *right-expr*
left-expr `>` *right-expr*
left-expr `<=` *right-expr*

left-expr `>=` *right-expr*

`<` denotes less than, `<=` less than or equal, `>` greater than, and `>=` greater than or equal of its two operands. For each type of objects the definition of the ordering is given in the respective chapter. The ordering of objects of different types is as follows. Rationals are smallest, next are cyclotomics, followed by finite field elements, permutations, words, words in solvable groups, boolean values, functions, lists, and records are largest.

Comparison operators, which includes the operator `in` (see 3.13) are not associative, i.e., it is not allowed to write *a* `=` *b* `<>` *c* `=` *d*, you must use `(`*a* `=` *b*`) <> (`*c* `=` *d*`)` instead. The comparison operators have higher precedence than the logical operators, but lower precedence than the arithmetic operators (see 2.10). Thus, for example, *a* `*` *b* `=` *c* `and` *d* is interpreted, `((`*a* `*` *b*`) =` *c*`) and` *d*`)`.

```
gap> 2 * 2 + 9 = Fibonacci(7);      # a comparison where the left
true                                # operand is an expression
```

2.10 Operations

`+` *right-expr*
`-` *right-expr*
left-expr `+` *right-expr*
left-expr `-` *right-expr*
left-expr `*` *right-expr*
left-expr `/` *right-expr*
left-expr `mod` *right-expr*
left-expr `^` *right-expr*

The arithmetic operators are `+`, `-`, `*`, `/`, `mod`, and `^`. The meanings (semantic) of those operators generally depend on the types of the operands involved, and they are defined in the various chapters describing the types. However basically the meanings are as follows.

`+` denotes the addition, and `-` the subtraction of ring and field elements. `*` is the multiplication of group elements, `/` is the multiplication of the left operand with the inverse of the right operand. `mod` is only defined for integers and rationals and denotes the modulo operation. `+` and `-` can also be used as unary operations. The unary `+` is ignored and unary `-` is equivalent to multiplication by -1. `^` denotes powering of a group element if the right operand is an integer, and is also used to denote operation if the right operand is a group element.

The **precedence** of those operators is as follows. The powering operator `^` has the highest precedence, followed by the unary operators `+` and `-`, which are followed by the multiplicative operators `*`, `/`, and `mod`, and the additive binary operators `+` and `-` have the lowest precedence. That means that the expression `-2 ^ -2 * 3 + 1` is interpreted as `(-(2 ^ (-2)) * 3) + 1`. If in doubt use parentheses to clarify your intention.

The **associativity** of the arithmetic operators is as follows.`^` is not associative, i.e., it is illegal to write `2^3^4`, use parentheses to clarify whether you mean `(2^3) ^ 4` or `2 ^ (3^4)`. The unary operators `+` and `-` are right associative, because they are written to the left of their operands. `*`, `/`, `mod`, `+`, and `-` are all left associative, i.e., `1-2-3` is interpreted as `(1-2)-3` not as `1-(2-3)`. Again, if in doubt use parentheses to clarify your intentions.

The arithmetic operators have higher precedence than the comparison operators (see 2.9 and 3.13) and the logical operators. Thus, for example, *a* `*` *b* `=` *c* `and` *d* is interpreted, `((`*a* `*` *b*`) =` *c*`) and` *d*.

```
gap> 2 * 2 + 9;    # a very simple arithmetic expression
13
```

2.11 Statements

Assignments (see 2.12), Procedure calls (see 2.13), `if` statements (see 2.14), `while` (see 2.15), `repeat` (see 2.16) and `for` loops (see 2.17), and the `return` statement (see 2.19) are called statements. They can be entered interactively or be part of a function definition. Every statement must be terminated by a semicolon.

Statements, unlike expressions, have no value. They are executed only to produce an effect. For example an assignment has the effect of assigning a value to a variable, a `for` loop has the effect of executing a statement sequence for all elements in a list and so on. We will talk about **evaluation** of expressions but about **execution** of statements to emphasize this difference.

It is possible to use expressions as statements. However this does cause a warning.

```
gap> if i <> 0  then  k = 16/i;  fi;
Syntax error: warning, this statement has no effect
if i <> 0  then  k = 16/i;  fi;
                          ^
```

As you can see from the example this is useful for those users who are used to languages where `=` instead of `:=` denotes assignment.

A sequence of one or more statements is a statement sequence, and may occur everywhere instead of a single statement. There is nothing like PASCAL's BEGIN-END, instead each construct is terminated by a keyword. The most simple statement sequence is a single semicolon, which can be used as an empty statement sequence.

2.12 Assignments

var `:=` *expr*`;`

The **assignment** has the effect of assigning the value of the expressions *expr* to the variable *var*.

The variable *var* may be an ordinary variable (see 2.7), a list element selection *list-var*`[`*int-expr*`]` (see 3.5) or a record component selection *record-var*`.`*ident* (see 5.2). Since a list element or a record component may itself be a list or a record the left hand side of an assignment may be arbitrarily complex.

Note that variables do not have a type. Thus any value may be assigned to any variable. For example a variable with an integer value may be assigned a permutation or a list or anything else.

If the expression *expr* is a function call then this function must return a value. If the function does not return a value an error is signalled and you enter a break loop. As usual you can leave the break loop with `quit;`. If you enter `return` *return-expr*`;` the value of the expression *return-expr* is assigned to the variable, and execution continues after the assignment.

```
gap> S6 := rec( size := 720 );; S6;
rec(
  size := 720 )
gap> S6.generators := [ (1,2), (1,2,3,4,5) ];; S6;
rec(
  size := 720,
  generators := [ (1,2), (1,2,3,4,5) ] )
gap> S6.generators[2] := (1,2,3,4,5,6);; S6;
rec(
  size := 720,
  generators := [ (1,2), (1,2,3,4,5,6) ] )
```

2.13 Procedure Calls

procedure-var`();`
procedure-var`(` *arg-expr* {`,` *arg-expr*} `);`

The procedure call has the effect of calling the procedure *procedure-var*. A procedure call is done exactly like a function call (see 2.8). The distinction between functions and procedures is only for the sake of the discussion, GAP does not distinguish between them.

A **function** does return a value but does not produce a side effect. As a convention the name of a function is a noun, denoting what the function returns, e.g., `Length`, `Concatenation` and `Order`.

A **procedure** is a function that does not return a value but produces some effect. Procedures are called only for this effect. As a convention the name of a procedure is a verb, denoting what the procedure does, e.g., `Print`, `Append` and `Sort`.

```
gap> Read( "myfile.g" );     # a call to the procedure Read
gap> l := [ 1, 2 ];;
gap> Append( l, [3,4,5] );    # a call to the procedure Append
```

2.14 If

`if` *bool-expr1* `then` *statements1*
{ `elif` *bool-expr2* `then` *statements2* }
[`else` *statements3*]
`fi;`

The `if` statement allows one to execute statements depending on the value of some boolean expression. The execution is done as follows.

First the expression *bool-expr1* following the `if` is evaluated. If it evaluates to `true` the

statement sequence *statements1* after the first `then` is executed, and the execution of the `if` statement is complete.

Otherwise the expressions *bool-expr2* following the `elif` are evaluated in turn. There may be any number of `elif` parts, possibly none at all. As soon as an expression evaluates to `true` the corresponding statement sequence *statements2* is executed and execution of the `if` statement is complete.

If the `if` expression and all, if any, `elif` expressions evaluate to `false` and there is an `else` part, which is optional, its statement sequence *statements3* is executed and the execution of the `if` statement is complete. If there is no `else` part the `if` statement is complete without executing any statement sequence.

Since the `if` statement is terminated by the `fi` keyword there is no question where an `else` part belongs, i.e., GAP has no dangling else.
In `if` *expr1* `then if` *expr2* `then` *stats1* `else` *stats2* `fi; fi;`
the `else` part belongs to the second `if` statement, whereas in
`if` *expr1* `then if` *expr2* `then` *stats1* `fi; else` *stats2* `fi;`
the `else` part belongs to the first `if` statement.

Since an if statement is not an expression it is not possible to write

```
abs := if x  > 0  then  x;  else  -x;  fi;
```

which would, even if legal syntax, be meaningless, since the `if` statement does not produce a value that could be assigned to `abs`.

If one expression evaluates neither to `true` nor to `false` an error is signalled and a break loop is entered. As usual you can leave the break loop with `quit;`. If you enter `return true;`, execution of the `if` statement continues as if the expression whose evaluation failed had evaluated to `true`. Likewise, if you enter `return false;`, execution of the `if` statement continues as if the expression whose evaluation failed had evaluated to `false`.

```
gap> i := 10;;
gap> if 0 < i  then
>        s := 1;
>    elif i < 0  then
>        s := -1;
>    else
>        s := 0;
>    fi;
gap> s;
1        #  the sign of i
```

2.15 While

`while` *bool-expr* `do` *statements* `od;`

The `while` loop executes the statement sequence *statements* while the condition *bool-expr* evaluates to `true`.

First *bool-expr* is evaluated. If it evaluates to `false` execution of the `while` loop terminates

and the statement immediately following the `while` loop is executed next. Otherwise if it evaluates to `true` the *statements* are executed and the whole process begins again.

The difference between the `while` loop and the `repeat until` loop (see 2.16) is that the *statements* in the `repeat until` loop are executed at least once, while the *statements* in the `while` loop are not executed at all if *bool-expr* is `false` at the first iteration.

If *bool-expr* does not evaluate to `true` or `false` an error is signalled and a break loop is entered. As usual you can leave the break loop with `quit;`. If you enter `return false;`, execution continues with the next statement immediately following the `while` loop. If you enter `return true;`, execution continues at *statements*, after which the next evaluation of *bool-expr* may cause another error.

```
gap> i := 0;;  s := 0;;
gap> while s <= 200  do
>        i := i + 1;  s := s + i^2;
>    od;
gap> s;
204         #  first sum of the first i squares larger than 200
```

2.16 Repeat

`repeat` *statements* `until` *bool-expr*`;`

The `repeat` loop executes the statement sequence *statements* until the condition *bool-expr* evaluates to `true`.

First *statements* are executed. Then *bool-expr* is evaluated. If it evaluates to `true` the `repeat` loop terminates and the statement immediately following the `repeat` loop is executed next. Otherwise if it evaluates to `false` the whole process begins again with the execution of the *statements*.

The difference between the `while` loop (see 2.15) and the `repeat until` loop is that the *statements* in the `repeat until` loop are executed at least once, while the *statements* in the `while` loop are not executed at all if *bool-expr* is `false` at the first iteration.

If *bool-expr* does not evaluate to `true` or `false` a error is signalled and a break loop is entered. As usual you can leave the break loop with `quit;`. If you enter `return true;`, execution continues with the next statement immediately following the `repeat` loop. If you enter `return false;`, execution continues at *statements*, after which the next evaluation of *bool-expr* may cause another error.

```
gap> i := 0;;  s := 0;;
gap> repeat
>        i := i + 1;  s := s + i^2;
>    until s > 200;
gap> s;
204         #  first sum of the first i squares larger than 200
```

2.17 For

`for` *simple-var* `in` *list-expr* `do` *statements* `od;`

The `for` loop executes the statement sequence *statements* for every element of the list *list-expr*.

The statement sequence *statements* is first executed with *simple-var* bound to the first element of the list *list*, then with *simple-var* bound to the second element of *list* and so on. *simple-var* must be a simple variable, it must not be a list element selection *list-var*`[`*int-expr*`]` or a record component selection *record-var*`.`*ident*.

The execution of the `for` loop is exactly equivalent to the `while` loop

```
loop-list := list;
loop-index := 1;
while loop-index <= Length(loop-list) do
    variable := loop-list[loop-index];
    statements
    loop-index := loop-index + 1;
od;
```

with the exception that *loop-list* and *loop-index* are different variables for each `for` loop that do not interfere with each other.

The list *list* is very often a range.
`for` *variable* `in [`*from*`..`*to*`] do` *statements* `od;`
corresponds to the more common
`for` *variable* `from` *from* `to` *to* `do` *statements* `od;`
in other programming languages.

```
gap> s := 0;;
gap> for i  in [1..100]  do
>        s := s + i;
> od;
gap> s;
5050
```

Note in the following example how the modification of the **list** in the loop body causes the loop body also to be executed for the new values

```
gap> l := [ 1, 2, 3, 4, 5, 6 ];;
gap> for i  in l  do
>        Print( i, " " );
>        if i mod 2 = 0  then Add( l, 3 * i / 2 );  fi;
> od;  Print( "\n" );
1 2 3 4 5 6 3 6 9 9
gap> l;
[ 1, 2, 3, 4, 5, 6, 3, 6, 9, 9 ]
```

Note in the following example that the modification of the **variable** that holds the list has no influence on the loop

```
gap> l := [ 1, 2, 3, 4, 5, 6 ];;
```

```
gap> for i  in l  do
>        Print( i, " " );
>        l := [];
> od;  Print( "\n" );
1 2 3 4 5 6
gap> l;
[  ]
```

2.18 Functions

`function` ([*arg-ident* {`,` *arg-ident*}])
    [`local` *loc-ident* {`,` *loc-ident*} `;`]
    statements
`end`

A function is in fact a literal and not a statement. Such a function literal can be assigned to a variable or to a list element or a record component. Later this function can be called as described in 2.8.

The following is an example of a function definition. It is a function to compute values of the Fibonacci sequence

```
gap> fib := function ( n )
>         local  f1,  f2,  f3,  i;
>         f1 := 1;  f2 := 1;
>         for i  in [3..n]  do
>             f3 := f1 + f2;
>             f1 := f2;
>             f2 := f3;
>         od;
>         return f2;
>     end;;
gap> List( [1..10], fib );
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

Because for each of the formal arguments *arg-ident* and for each of the formal locals *loc-ident* a new variable is allocated when the function is called (see 2.8), it is possible that a function calls itself. This is usually called **recursion**. The following is a recursive function that computes values of the Fibonacci sequence

```
gap> fib := function ( n )
>         if n < 3  then
>             return 1;
>         else
>             return fib(n-1) + fib(n-2);
>         fi;
>     end;;
gap> List( [1..10], fib );
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

Note that the recursive version needs `2 * fib(`*n*`)-1` steps to compute `fib(`*n*`)`, while the iterative version of `fib` needs only *n*`-2` steps. Both are not optimal however, the library function `Fibonacci` only needs on the order of `Log(`*n*`)` steps.

arg-ident `->` *expr*

This is a shorthand for
`function ( `*arg-ident*` ) return `*expr*`; end`.
arg-ident must be a single identifier, i.e., it is not possible to write functions of several arguments this way. Also `arg` is not treated specially, so it is also impossible to write functions that take a variable number of arguments this way.

The following is an example of a typical use of such a function

```
gap> Sum( List( [1..100], x -> x^2 ) );
338350
```

When a function *fun1* definition is evaluated inside another function *fun2*, GAP binds all the identifiers inside the function *fun1* that are identifiers of an argument or a local of *fun2* to the corresponding variable. This set of bindings is called the environment of the function *fun1*. When *fun1* is called, its body is executed in this environment. The following implementation of a simple stack uses this. Values can be pushed onto the stack and then later be popped off again. The interesting thing here is that the functions `push` and `pop` in the record returned by `Stack` access the local variable `stack` of `Stack`. When `Stack` is called a new variable for the identifier `stack` is created. When the function definitions of `push` and `pop` are then evaluated (as part of the `return` statement) each reference to `stack` is bound to this new variable. Note also that the two stacks `A` and `B` do not interfere, because each call of `Stack` creates a new variable for `stack`.

```
gap> Stack := function ()
>         local   stack;
>         stack := [];
>         return rec(
>             push := function ( value )
>                 Add( stack, value );
>             end,
>             pop := function ()
>                 local   value;
>                 value := stack[Length(stack)];
>                 Unbind( stack[Length(stack)] );
>                 return value;
>             end
>         );
>    end;;
gap> A := Stack();;
gap> B := Stack();;
gap> A.push( 1 );  A.push( 2 );  A.push( 3 );
gap> B.push( 4 );  B.push( 5 );  B.push( 6 );
gap> A.pop();  A.pop();  A.pop();
3
```

```
2
1
gap> B.pop();  B.pop();  B.pop();
6
5
4
```

This feature should be used rarely, since its implementation in GAP is not very efficient.

2.19 Return

`return;`

In this form `return` terminates the call of the innermost function that is currently executing, and control returns to the calling function. An error is signalled if no function is currently executing. No value is returned by the function.

`return` *expr*`;`

In this form `return` terminates the call of the innermost function that is currently executing, and returns the value of the expression *expr*. Control returns to the calling function. An error is signalled if no function is currently executing.

Both statements can also be used in break loops. `return;` has the effect that the computation continues where it was interrupted by an error or the user hitting *ctr*`C`. `return` *expr*`;` can be used to continue execution after an error. What happens with the value *expr* depends on the particular error.

2.20 The Syntax in BNF

This section contains the definition of the GAP syntax in Backus-Naur form.

A BNF is a set of rules, whose left side is the name of a syntactical construct. Those names are enclosed in angle brackets and written in *italics*. The right side of each rule contains a possible form for that syntactic construct. Each right side may contain names of other syntactic constructs, again enclosed in angle brackets and written in *italics*, or character sequences that must occur literally; they are written in `typewriter style`.

Furthermore each righthand side can contain the following metasymbols written in **boldface**. If the right hand side contains forms separated by a pipe symbol (|) this means that one of the possible forms can occur. If a part of a form is enclosed in square brackets ([ ]) this means that this part is optional, i.e. might be present or missing. If part of the form is enclosed in curly braces ({ }) this means that the part may occur arbitrarily often, or possibly be missing.

Ident := a|...|z|A|...|Z|_ {a|...|z|A|...|Z|0|...|9|_}
Var := *Ident*
| *Var* . *Ident*
| *Var* . (*Expr*)
| *Var* [*Expr*]
| *Var* { *Expr* }
| *Var* ([*Expr* { , *Expr* }])
List := [[*Expr*] {, [*Expr*] }]
| [*Expr* [, *Expr*] .. *Expr*]
Record := rec([*Ident* := *Expr* {, *Ident* := *Expr* }])
Permutation := (*Expr* {, *Expr* }) { (*Expr* {, *Expr* }) }
Function := function ([*Ident* {, *Ident* }])
[local *Ident* {, *Ident* } ;]
Statements
end
Char := ' *any character* '
String := " { *any character* } "
Int := 0|1|...|9 { 0|1|...|9 }
Atom := *Int*
| *Var*
| (*Expr*)
| *Permutation*
| *Char*
| *String*
| *Function*
| *List*
| *Record*
Factor := {+|-} *Atom* [^ {+|-} *Atom*]
Term := *Factor* { *|/|mod *Factor* }
Arith := *Term* { +|- *Term* }
Rel := { not } *Arith* { =|<>|<|>|<=|>=|in *Arith* }
And := *Rel* { and *Rel* }
Log := *And* { or *And* }
Expr := *Log*
| *Var* [-> *Log*]
Statement := *Expr*
| *Var* := *Expr*
| if *Expr* then *Statements*
{ elif *Expr* then *Statements* }
[else *Statements*] fi
| for *Var* in *Expr* do *Statements* od
| while *Expr* do *Statements* od
| repeat *Statements* until *Expr*
| return [*Expr*]
| quit
Statements := { *Statement* ; }
| ;

Chapter 3

Lists

Lists are the most important way to collect objects and treat them together. A **list** is a collection of elements. A list also implies a partial mapping from the integers to the elements. I.e., there is a first element of a list, a second, a third, and so on.

List constants are written by writing down the elements in order between square brackets `[`, `]`, and separating them with commas `,`. An **empty list**, i.e., a list with no elements, is written as `[]`.

```
gap> [ 1, 2, 3 ];
[ 1, 2, 3 ]    # a list with three elements
gap> [ [], [ 1 ], [ 1, 2 ] ];
[ [  ], [ 1 ], [ 1, 2 ] ]    # a list may contain other lists
```

Usually a list has no holes, i.e., contain an element at every position. However, it is absolutely legal to have lists with holes. They are created by leaving the entry between the commas empty. Lists with holes are sometimes convenient when the list represents a mapping from a finite, but not consecutive, subset of the positive integers. We say that a list that has no holes is **dense**.

```
gap> l := [ , 4, 9,, 25,, 49,,,, 121 ];;
gap> l[3];
9
gap> l[4];
Error, List Element: <list>[4] must have a value
```

It is most common that a list contains only elements of one type. This is not a must though. It is absolutely possible to have lists whose elements are of different types. We say that a list whose elements are all of the same type is **homogeneous**.

```
gap> l := [ 1, E(2), Z(3), (1,2,3), [1,2,3], "What a mess" ];;
gap> l[1];  l[3];  l[5][2];
1
Z(3)
2
```

The first sections describe the functions that test if an object is a list and convert an object to a list (see 3.1 and 3.2).

The next section describes how one can access elements of a list (see 3.3 and 3.4).

The next sections describe how one can change lists (see 3.5, 3.6, 3.7, 3.8, 3.10).

The next sections describe the operations applicable to lists (see 3.11 and 3.12).

The next sections describe how one can find elements in a list (see 3.13, 3.14, 3.15, 3.16).

The next sections describe the functions that construct new lists, e.g., sublists (see 3.17, 3.18, 3.19, 3.20, 3.21).

The next sections describe the functions deal with the subset of elements of a list that have a certain property (see 3.22, 3.23, 3.24, 3.25, 3.26, 3.27).

The next sections describe the functions that sort lists (see 3.28, 3.29, 3.30, 3.31).

The next sections describe the functions to compute the product, sum, maximum, and minimum of the elements in a list (see 3.32, 3.33, 3.34, 3.35, 3.36).

The final section describes the function that takes a random element from a list (see 3.37).

Lists are also used to represent sets, subsets, vectors, and ranges.

3.1 IsList

`IsList(` *obj* `)`

`IsList` returns `true` if the argument *obj*, which can be an arbitrary object, is a list and `false` otherwise. Will signal an error if *obj* is an unbound variable.

```
gap> IsList( [ 1, 3, 5, 7 ] );
true
gap> IsList( 1 );
false
```

3.2 List

`List(` *obj* `)`
`List(` *list*`,` *func* `)`

In its first form `List` returns the argument *obj*, which must be a list, a permutation, a string or a word, converted into a list. If *obj* is a list, it is simply returned. If *obj* is a permutation, `List` returns a list where the i-th element is the image of i under the permutation *obj*. If *obj* is a word, `List` returns a list where the i-th element is the i-th generator of the word, as a word of length 1.

```
gap> List( [1,2,3] );
[ 1, 2, 3 ]
gap> List( (1,2)(3,4,5) );
[ 2, 1, 4, 5, 3 ]
```

In its second form `List` returns a new list, where each element is the result of applying the function *func*, which must take exactly one argument and handle the elements of *list*, to the corresponding element of the list *list*. The list *list* must not contain holes.

```
gap> List( [1,2,3], x->x^2 );
[ 1, 4, 9 ]
gap> List( [1..10], IsPrime );
[ false, true, true, false, true, false, true, false, false, false ]
```

Note that this function is called `map` in Lisp and many other similar programming languages. This name violates the GAP rule that verbs are used for functions that change their arguments. According to this rule `map` would change *list*, replacing every element with the result of the application *func* to this argument.

3.3 List Elements

list`[` *pos* `]`

The above construct evaluates to the *pos*-th element of the list *list*. *pos* must be a positive integer. List indexing is done with origin 1, i.e., the first element of the list is the element at position 1.

```
gap> l := [ 2, 3, 5, 7, 11, 13 ];;
gap> l[1];
2
gap> l[2];
3
gap> l[6];
13
```

If *list* does not evaluate to a list, or *pos* does not evaluate to a positive integer, or *list*`[`*pos*`]` is unbound an error is signalled. As usual you can leave the break loop with `quit;`. On the other hand you can return a result to be used in place of the list element by `return` *expr*`;`.

list`{` *poss* `}`

The above construct evaluates to a new list *new* whose first element is *list*`[`*poss*`[1]]`, whose second element is *list*`[`*poss*`[2]]`, and so on. *poss* must be a dense list of positive integers, it need, however, not be sorted and may contain duplicate elements. If for any i, *list*`[` *poss*`[`i`]` `]` is unbound, an error is signalled.

```
gap> l := [ 2, 3, 5, 7, 11, 13, 17, 19 ];;
gap> l{[4..6]};
[ 7, 11, 13 ]
gap> l{[1,7,1,8]};
[ 2, 17, 2, 19 ]
```

The result is a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the left operand (see 3.8).

It is possible to nest such sublist extractions, as can be seen in the following example.

```
gap> m := [ [1,2,3], [4,5,6], [7,8,9], [10,11,12] ];;
gap> m{[1,2,3]}{[3,2]};
[ [ 3, 2 ], [ 6, 5 ], [ 9, 8 ] ]
gap> l := m{[1,2,3]};; l{[3,2]};
```

```
[ [ 7, 8, 9 ], [ 4, 5, 6 ] ]
```

Note the difference between the two examples. The latter extracts elements 1, 2, and 3 from m and then extracts the elements 3 and 2 from **this list**. The former extracts elements 1, 2, and 3 from m and then extracts the elements 3 and 2 from **each of those element lists**.

To be precise. With each selector [*pos*] or {*poss*} we associate a **level** that is defined as the number of selectors of the form {*poss*} to its left in the same expression. For example

```
      l[pos1]{poss2}{poss3}[pos4]{poss5}[pos6]
level   0      0      1     1      1     2
```

Then a selector *list*[*pos*] of level *level* is computed as `ListElement(`*list*`,`*pos*`,`*level*`)`, where `ListElement` is defined as follows

```
ListElement := function ( list, pos, level )
    if level = 0  then
        return list[pos];
    else
        return List( list, elm -> ListElement(elm,pos,level-1) );
    fi;
end;
```

and a selector *list*{*poss*} of level *level* is computed as `ListElements(`*list*`,`*poss*`,`*level*`)`, where `ListElements` is defined as follows

```
ListElements := function ( list, poss, level )
    if level = 0  then
        return list{poss};
    else
        return List( list, elm -> ListElements(elm,poss,level-1) );
    fi;
end;
```

3.4 Length

`Length(` *list* `)`

`Length` returns the length of the list *list*. The **length** is defined as 0 for the empty list, and as the largest positive integer *index* such that *list*[*index*] has an assigned value for nonempty lists. Note that the length of a list may change if new elements are added to it or assigned to previously unassigned positions.

```
gap> Length( [] );
0
gap> Length( [ 2, 3, 5, 7, 11, 13, 17, 19 ] );
8
gap> Length( [ 1, 2,,, 5 ] );
5
```

For lists that contain no holes `Length`, `Number` (see 3.22), and `Size` return the same value. For lists with holes `Length` returns the largest index of a bound entry, `Number` returns the

number of bound entries, and `Size` signals an error.

3.5 List Assignment

list`[` *pos* `] :=` *object*`;`

The list assignment assigns the object *object*, which can be of any type, to the list entry at the position *pos*, which must be a positive integer, in the list *list*. That means that accessing the *pos*-th element of the list *list* will return *object* after this assignment.

```
gap> l := [ 1, 2, 3 ];;
gap> l[1] := 3;;  l;                # assign a new object
[ 3, 2, 3 ]
gap> l[2] := [ 4, 5, 6 ];;  l;      # object may be of any type
[ 3, [ 4, 5, 6 ], 3 ]
gap> l[ l[1] ] := 10;;  l;          # index may be an expression
[ 3, [ 4, 5, 6 ], 10 ]
```

If the index *pos* is larger than the length of the list *list* (see 3.4), the list is automatically enlarged to make room for the new element. Note that it is possible to generate lists with holes that way.

```
gap> l[4] := "another entry";;  l; # list is enlarged
[ 3, [ 4, 5, 6 ], 10, "another entry" ]
gap> l[ 10 ] := 1;;  l;             # now list has a hole
[ 3, [ 4, 5, 6 ], 10, "another entry",,,,,, 1 ]
```

The function `Add` (see 3.6) should be used if you want to add an element to the end of the list.

Note that assigning to a list changes the list. The ability to change an object is only available for lists and records (see 3.8).

If *list* does not evaluate to a list, *pos* does not evaluate to a positive integer or *object* is a call to a function which does not return a value, for example `Print`, an error is signalled As usual you can leave the break loop with `quit;`. On the other hand you can continue the assignment by returning a list, an index or an object using `return` *expr*`;`.

list`{` *poss* `} :=` *objects*`;`

The list assignment assigns the object *objects*`[1]`, which can be of any type, to the list *list* at the position *poss*`[1]`, the object *objects*`[2]` to *list*`[`*poss*`[2]]`, and so on. *poss* must be a dense list of positive integers, it need, however, not be sorted and may contain duplicate elements. *objects* must be a dense list and must have the same length as *poss*.

```
gap> l := [ 2, 3, 5, 7, 11, 13, 17, 19 ];;
gap> l{[1..4]} := [10..13];;  l;
[ 10, 11, 12, 13, 11, 13, 17, 19 ]
gap> l{[1,7,1,10]} := [ 1, 2, 3, 4 ];; l;
[ 3, 11, 12, 13, 11, 13, 2, 19,, 4 ]
```

It is possible to nest such sublist assignments, as can be seen in the following example.

```
gap> m := [ [1,2,3], [4,5,6], [7,8,9], [10,11,12] ];;
gap> m{[1,2,3]}{[3,2]} := [ [11,12], [13,14], [15,16] ];;  m;
[ [ 1, 12, 11 ], [ 4, 14, 13 ], [ 7, 16, 15 ], [ 10, 11, 12 ] ]
```

The exact behaviour is defined in the same way as for list extractions (see 3.3). Namely with each selector `[`*pos*`]` or {*poss*} we associate a **level** that is defined as the number of selectors of the form {*poss*} to its left in the same expression. For example

```
      l[pos1]{poss2}{poss3}[pos4]{poss5}[pos6]
level   0      0      1     1      1     2
```

Then a list assignment *list*`[`*pos*`] :=` *vals*`;` of level *level* is computed as `ListAssignment(` *list*`,` *pos*`,` *vals*`,` *level* `)`, where `ListAssignment` is defined as follows

```
ListAssignment := function ( list, pos, vals, level )
    local  i;
    if level = 0  then
        list[pos] := vals;
    else
        for i  in [1..Length(list)]  do
            ListAssignment( list[i], pos, vals[i], level-1 );
        od;
    fi;
end;
```

and a list assignment *list*{*poss*} `:=` *vals* of level *level* is computed as `ListAssignments(` *list*`,` *poss*`,` *vals*`,` *level* `)`, where `ListAssignments` is defined as follows

```
ListAssignments := function ( list, poss, vals, level )
    local  i;
    if level = 0  then
        list{poss} := vals;
    else
        for i  in [1..Length(list)]  do
            ListAssignments( list[i], poss, vals[i], level-1 );
        od;
    fi;
end;
```

3.6 Add

`Add(` *list*`,` *elm* `)`

`Add` adds the element *elm* to the end of the list *list*, i.e., it is equivalent to the assignment *list*`[ Length(`*list*`) + 1 ] :=` *elm*. The list is automatically enlarged to make room for the new element. `Add` returns nothing, it is called only for its side effect.

Note that adding to a list changes the list. The ability to change an object is only available for lists and records (see 3.8).

To add more than one element to a list use `Append` (see 3.7).

```
gap> l := [ 2, 3, 5 ];;  Add( l, 7 );  l;
[ 2, 3, 5, 7 ]
```

3.7 Append

`Append( list1, list2 )`

`Append` adds (see 3.6) the elements of the list *list2* to the end of the list *list1*. *list2* may contain holes, in which case the corresponding entries in *list1* will be left unbound. `Append` returns nothing, it is called only for its side effect.

```
gap> l := [ 2, 3, 5 ];; Append( l, [ 7, 11, 13 ] );  l;
[ 2, 3, 5, 7, 11, 13 ]
gap> Append( l, [ 17,, 23 ] ); l;
[ 2, 3, 5, 7, 11, 13, 17,, 23 ]
```

Note that appending to a list changes the list. The ability to change an object is only available for lists and records (see 3.8).

Note that `Append` changes the first argument, while `Concatenation` (see 3.17) creates a new list and leaves its arguments unchanged. As usual the name of the function that work destructively is a verb, but the name of the function that creates a new object is a substantive.

3.8 Identical Lists

With the list assignment (see 3.5, 3.6, 3.7) it is possible to change a list. The ability to change an object is only available for lists and records. This section describes the semantic consequences of this fact.

You may think that in the following example the second assignment changes the integer, and that therefore the above sentence, which claimed that only lists and records can be changed is wrong

```
i := 3;
i :=  i + 1;
```

But in this example not the **integer** `3` is changed by adding one to it. Instead the **variable** `i` is changed by assigning the value of `i+1`, which happens to be `4`, to `i`. The same thing happens in the following example

```
l := [ 1,  2 ];
l := [ 1, 2,  3 ];
```

The second assignment does not change the first list, instead it assigns a new list to the variable `l`. On the other hand, in the following example the list is changed by the second assignment.

```
l := [ 1,  2 ];
l[3] := 3;
```

To understand the difference first think of a variable as a name for an object. The important point is that a list can have several names at the same time. An assignment *var* `:=` *list*`;`

means in this interpretation that *var* is a name for the object *list*. At the end of the following example `l2` still has the value `[ 1, 2 ]` as this list has not been changed and nothing else has been assigned to it.

```
l1 := [ 1, 2 ];
l2 := l1;
l1 := [ 1, 2, 3 ];
```

But after the following example the list for which `l2` is a name has been changed and thus the value of `l2` is now `[ 1, 2, 3 ]`.

```
l1 := [ 1, 2 ];
l2 := l1;
l1[3] := 3;
```

We shall say that two lists are **identical** if changing one of them by a list assignment also changes the other one. This is slightly incorrect, because if **two** lists are identical, there are actually only two names for **one** list. However, the correct usage would be very awkward and would only add to the confusion. Note that two identical lists must be equal, because there is only one list with two different names. Thus identity is an equivalence relation that is a refinement of equality.

Let us now consider under which circumstances two lists are identical.

If you enter a list literal than the list denoted by this literal is a new list that is not identical to any other list. Thus in the following example `l1` and `l2` are not identical, though they are equal of course.

```
l1 := [ 1, 2 ];
l2 := [ 1, 2 ];
```

Also in the following example, no lists in the list `l` are identical.

```
l := [];
for i  in [1..10]  do l[i] := [ 1, 2 ];  od;
```

If you assign a list to a variable no new list is created. Thus the list value of the variable on the left hand side and the list on the right hand side of the assignment are identical. So in the following example `l1` and `l2` are identical lists.

```
l1 := [ 1, 2 ];
l2 := l1;
```

If you pass a list as argument, the old list and the argument of the function are identical. Also if you return a list from a function, the old list and the value of the function call are identical. So in the following example `l1` and `l2` are identical list

```
l1 := [ 1, 2 ];
f := function ( l )  return l;  end;
l2 := f( l1 );
```

The functions `Copy` and `ShallowCopy` (see 5.11 and 5.12) accept a list and return a new list that is equal to the old list but that is **not** identical to the old list. The difference between `Copy` and `ShallowCopy` is that in the case of `ShallowCopy` the corresponding elements of the new and the old lists will be identical, whereas in the case of `Copy` they will only be equal. So in the following example `l1` and `l2` are not identical lists.

```
l1 := [ 1, 2 ];
l2 := Copy( l1 );
```

If you change a list it keeps its identity. Thus if two lists are identical and you change one of them, you also change the other, and they are still identical afterwards. On the other hand, two lists that are not identical will never become identical if you change one of them. So in the following example both `l1` and `l2` are changed, and are still identical.

```
l1 := [ 1, 2 ];
l2 := l1;
l1[1] := 2;
```

3.9 IsIdentical

`IsIdentical( `*l*`, `*r*` )`

`IsIdentical` returns `true` if the objects *l* and *r* are identical. Unchangeable objects are considered identical if the are equal. Changeable objects, i.e., lists and records, are identical if changing one of them by an assignment also changes the other one, as described in 3.8.

```
gap> IsIdentical( 1, 1 );
true
gap> IsIdentical( 1, () );
false
gap> l := [ 'h', 'a', 'l', 'l', 'o' ];;
gap> l = "hallo";
true
gap> IsIdentical( l, "hallo" );
false
```

3.10 Enlarging Lists

The previous section (see 3.5) told you (among other things), that it is possible to assign beyond the logical end of a list, automatically enlarging the list. This section tells you how this is done.

It would be extremly wasteful to make all lists large enough so that there is room for all assignments, because some lists may have more than 100000 elements, while most lists have less than 10 elements.

On the other hand suppose every assignment beyond the end of a list would be done by allocating new space for the list and copying all entries to the new space. Then creating a list of 1000 elements by assigning them in order, would take half a million copy operations and also create a lot of garbage that the garbage collector would have to reclaim.

So the following strategy is used. If a list is created it is created with exactly the correct size. If a list is enlarged, because of an assignment beyond the end of the list, it is enlarged by at least *length*`/8 + 4` entries. Therefore the next assignments beyond the end of the list do not need to enlarge the list. For example creating a list of 1000 elements by assigning them in

order, would now take only 32 enlargements.

The result of this is of course that the **physical length**, which is also called the size, of a list may be different from the **logical length**, which is usually called simply the length of the list. Aside from the implications for the performance you need not be aware of the physical length. In fact all you can ever observe, for example by calling `Length` is the logical length.

Suppose that `Length` would have to take the physical length and then test how many entries at the end of a list are unassigned, to compute the logical length of the list. That would take too much time. In order to make `Length`, and other functions that need to know the logical length, more efficient, the length of a list is stored along with the list.

A note aside. In the previous version 2.4 of GAP a list was indeed enlarged every time an assignment beyond the end of the list was performed. To deal with the above inefficiency the following hacks where used. Instead of creating lists in order they were usually created in reverse order. In situations where this was not possible a dummy assignment to the last position was performed, for example

```
l := [];
l[1000] := "dummy";
l[1] := first_value();
for i  from 2  to 1000  do l[i] := next_value(l[i-1]);  od;
```

3.11 Comparisons of Lists

list1 `=` *list2*
list1 `<>` *list2*

The equality operator `=` evaluates to `true` if the two lists *list1* and *list2* are equal and `false` otherwise. The inequality operator `<>` evaluates to `true` if the two lists are not equal and `false` otherwise. Two lists *list1* and *list2* are equal if and only if for every index i, either both entries *list1*`[`i`]` and *list2*`[`i`]` are unbound, or both are bound and are equal, i.e., *list1*`[`i`]` `=` *list2*`[`i`]` is `true`.

```
gap> [ 1, 2, 3 ] = [ 1, 2, 3 ];
true
gap> [ , 2, 3 ] = [ 1, 2, ];
false
gap> [ 1, 2, 3 ] = [ 3, 2, 1 ];
false
```

list1 `<` *list2*, *list1* `<=` *list2* *list1* `>` *list2*, *list1* `>=` *list2*

The operators `<`, `<=`, `>` and `>=` evaluate to `true` if the list *list1* is less than, less than or equal to, greater than, or greater than or equal to the list *list2* and to `false` otherwise. Lists are ordered lexicographically, with unbound entries comparing very small. That means the following. Let i be the smallest positive integer i, such that neither both entries *list1*`[`i`]` and *list2*`[`i`]` are unbound, nor both are bound and equal. Then *list1* is less than *list2* if either *list1*`[`i`]` is unbound (and *list2*`[`i`]` is not) or both are bound and *list1*`[`i`]` `<` *list2*`[`i`]` is `true`.

```
gap> [ 1, 2, 3, 4 ] < [ 1, 2, 4, 8 ];
```

```
true     # list1[3] < list2[3]
gap> [ 1, 2, 3 ] < [ 1, 2, 3, 4 ];
true     # list1[4] is unbound and therefore very small
gap> [ 1, , 3, 4 ] < [ 1, 2, 3 ];
true     # list1[2] is unbound and therefore very small
```

You can also compare objects of other types, for example integers or permutations with lists. Of course those objects are never equal to a list. Records are greater than lists, objects of every other type are smaller than lists.

```
gap> 123 < [ 1, 2, 3 ];
true
gap> [ 1, 2, 3 ] < rec( a := 123 );
true
```

3.12 Operations for Lists

list `*` *obj*
obj `*` *list*

The operator `*` evaluates to the product of list *list* by an object *obj*. The product is a new list that at each position contains the product of the corresponding element of *list* by *obj*. *list* may contain holes, in which case the result will contain holes at the same positions.

The elements of *list* and *obj* must be objects of the following types; integers, rationals, cyclotomics, elements of a finite field, permutations, matrices, words in abstract generators, or words in solvable groups.

```
gap> [ 1, 2, 3 ] * 2;
[ 2, 4, 6 ]
gap> 2 * [ 2, 3,, 5,, 7 ];
[ 4, 6,, 10,, 14 ]
gap> [ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ] * (1,4);
[ (1,4), (1,4)(2,3), (1,2,4), (1,2,3,4), (1,3,2,4), (1,3,4) ]
```

Many more operators are available for vectors and matrices, which are also represented by lists.

3.13 In

elm `in` *list*

The `in` operator evaluates to `true` if the object *elm* is an element of the list *list* and to `false` otherwise. *elm* is an element of *list* if there is a positive integer *index* such that *list*`[`*index*`]=`*elm* is `true`. *elm* may be an object of an arbitrary type and *list* may be a list containing elements of any type.

It is much faster to test for membership for sets, because for sets, which are always sorted, `in` can use a binary search, instead of the linear search used for ordinary lists. So if you have a list for which you want to perform a large number of membership tests you may consider

converting it to a set with the function `Set` (see 4.2).

```
gap> 1 in [ 2, 2, 1, 3 ];
true
gap> 1 in [ 4, -1, 0, 3 ];
false
gap> s := Set([2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32]);;
gap> 17 in s;
false        # uses binary search and only 4 comparisons
gap> 1 in [ "This", "is", "a", "list", "of", "strings" ];
false
gap> [1,2] in [ [0,6], [0,4], [1,3], [1,5], [1,2], [3,4] ];
true
```

`Position` (see 3.14) and `PositionSorted` (see 3.15) allow you to find the position of an element in a list.

3.14 Position

`Position(` *list*`,` *elm* `)`

`Position` returns the position of the element *elm*, which may be an object of any type, in the list *list*. If the element is not in the list the result is `false`. If the element appears several times, the first position is returned.

It is much faster to search for an element in a set, because for sets, which are always sorted, `Position` can use a binary search, instead of the linear search used for ordinary lists. So if you have a list for which you want to perform a large number of searches you may consider converting it to a set with the function `Set` (see 4.2).

```
gap> Position( [ 2, 2, 1, 3 ],  1 );
3
gap> Position( [ 2, 1, 1, 3 ], 1 );
2
gap> Position( [ 4, -1, 0, 3 ],  1 );
false
gap> s := Set([2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32]);;
gap> Position( s,  17 );
false        # uses binary search and only 4 comparisons
gap> Position( [ "This", "is", "a", "list", "of", "strings" ],  1 );
false
gap> Position( [ [0,6], [0,4], [1,3], [1,5], [1,2], [3,4] ],  [1,2] );
5
```

The `in` operator (see 3.13) can be used if you are only interested to know whether the element is in the list or not. `PositionSorted` (see 3.15) can be used if the list is sorted. `PositionProperty` (see 3.16) allows you to find the position of an element that satisfies a certain property in a list.

3.15 PositionSorted

`PositionSorted(` *list*`,` *elm* `)`
`PositionSorted(` *list*`,` *elm*`,` *func* `)`

In the first form `PositionSorted` returns the position of the element *elm*, which may be an object of any type, with respect to the sorted list *list*.

In the second form `PositionSorted` returns the position of the element *elm*, which may be an object of any type with respect to the list *list*, which must be sorted with respect to *func*. *func* must be a function of two arguments that returns `true` if the first argument is less than the second argument and `false` otherwise.

`PositionSorted` returns *pos* such that *list*`[`*pos*`-1] <`*elm* and *elm* `<=` *list*`[`*pos*`]`. That means, if *elm* appears once in *list*, its position is returned. If *elm* appears several times in *list*, the position of the first occurrence is returned. If *elm* is not an element of *list*, the index where *elm* must be inserted to keep the list sorted is returned.

```
gap> PositionSorted( [1,4,5,5,6,7], 0 );
1
gap> PositionSorted( [1,4,5,5,6,7], 2 );
2
gap> PositionSorted( [1,4,5,5,6,7], 4 );
2
gap> PositionSorted( [1,4,5,5,6,7], 5 );
3
gap> PositionSorted( [1,4,5,5,6,7], 8 );
7
```

`Position` (see 3.14) is another function that returns the position of an element in a list. `Position` accepts unsorted lists, uses linear instead of binary search and returns `false` if *elm* is not in *list*.

3.16 PositionProperty

`PositionProperty(` *list*`,` *func* `)`

`PositionProperty` returns the position of the first element in the list *list* for which the unary function *func* returns `true`. *list* must not contain holes. If *func* returns `false` for all elements of *list* `false` is returned. *func* must return `true` or `false` for every element of *list*, otherwise an error is signalled.

```
gap> PositionProperty( [10^7..10^8], IsPrime );
20
gap> PositionProperty( [10^5..10^6],
>                      n -> not IsPrime(n) and IsPrimePowerInt(n) );
490
```

`First` (see 3.27) allows you to extract the first element of a list that satisfies a certain property.

3.17 Concatenation

`Concatenation(` *list1* `,` *list2* `..` `)`
`Concatenation(` *list* `)`

In the first form `Concatenation` returns the concatenation of the lists *list1*, *list2*, etc. The **concatenation** is the list that begins with the elements of *list1*, followed by the elements of *list2* and so on. Each list may also contain holes, in which case the concatenation also contains holes at the corresponding positions.

```
gap> Concatenation( [ 1, 2, 3 ], [ 4, 5 ] );
[ 1, 2, 3, 4, 5 ]
gap> Concatenation( [2,3,,5,,7], [11,,13,,,,17,,19] );
[ 2, 3,, 5,, 7, 11,, 13,,,, 17,, 19 ]
```

In the second form *list* must be a list of lists *list1*, *list2*, etc, and `Concatenation` returns the concatenation of those lists.

```
gap> Concatenation( [ [1,2,3], [2,3,4], [3,4,5] ] );
[ 1, 2, 3, 2, 3, 4, 3, 4, 5 ]
```

The result is a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the argument lists (see 3.8).

Note that `Concatenation` creates a new list and leaves it arguments unchanged, while `Append` (see 3.7) changes its first argument. As usual the name of the function that works destructively is a verb, but the name of the function that creates a new object is a substantive.

`Set(Concatenation(`*set1*`,`*set2*`..))` (see 4.2) is a way to compute the union of sets, however, `Union` is more efficient.

3.18 Flat

`Flat(` *list* `)`

`Flat` returns the list of all elements that are contained in the list *list* or its sublists. That is, `Flat` first makes a new empty list *new*. Then it loops over the elements *elm* of *list*. If *elm* is not a list it is added to *new*, otherwise `Flat` appends `Flat(` *elm* `)` to *new*.

```
gap> Flat( [ 1, [ 2, 3 ], [ [ 1, 2 ], 3 ] ] );
[ 1, 2, 3, 1, 2, 3 ]
gap> Flat( [ ] );
[ ]
```

3.19 Reversed

`Reversed(` *list* `)`

`Reversed` returns a new list that contains the elements of the list *list*, which must not contain holes, in reverse order. The argument list is unchanged.

```
gap> Reversed( [ 1, 4, 5, 5, 6, 7 ] );
```

```
[ 7, 6, 5, 5, 4, 1 ]
```

The result is a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the argument list (see 3.8).

3.20 Sublist

`Sublist(` *list*`,` *inds* `)`

`Sublist` returns a new list in which the i-th element is the element *list*`[` *inds*`[` i `]]`, of the list *list*. *inds* must be a list of positive integers without holes, it need, however, not be sorted and may contains duplicate elements. If *list*`[` *inds*`[` i `]]` is unbound for an i, an error is signalled.

```
gap> Sublist( [ 2, 3, 5, 7, 11, 13, 17, 19 ], [4..6] );
[ 7, 11, 13 ]
gap> Sublist( [ 2, 3, 5, 7, 11, 13, 17, 19 ], [1,7,1,8] );
[ 2, 17, 2, 19 ]
```

The result is a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the argument list (see 3.8).

`Filtered` (see 3.24) allows you to extract elements from a list according to a predicate.

`Sublist` has been made obsolete by the introduction of the construct *list*`{` *inds* `}` (see 3.3).

3.21 Cartesian

`Cartesian(` *list1*`,` *list2*`..` `)`
`Cartesian(` *list* `)`

In the first form `Cartesian` returns the cartesian product of the lists *list1*, *list2*, etc.

In the second form *list* must be a list of lists *list1*, *list2*, etc., and `Cartesian` returns the cartesian product of those lists.

The **cartesian product** is a list *cart* of lists *tup*, such that the first element of *tup* is an element of *list1*, the second element of *tup* is an element of *list2*, and so on. The total number of elements in *cart* is the product of the lengths of the argument lists. In particular *cart* is empty if and only if at least one of the argument lists is empty. Also *cart* contains duplicates if and only if no argument list is empty and at least one contains duplicates.

The last index runs fastest. That means that the first element *tup1* of *cart* contains the first element from *list1*, from *list2* and so on. The second element *tup2* of *cart* contains the first element from *list1*, the first from *list2*, an so on, but the last element of *tup2* is the second element of the last argument list. This implies that *cart* is a set if and only if all argument lists are sets.

```
gap> Cartesian( [1,2], [3,4], [5,6] );
[ [ 1, 3, 5 ], [ 1, 3, 6 ], [ 1, 4, 5 ], [ 1, 4, 6 ], [ 2, 3, 5 ],
  [ 2, 3, 6 ], [ 2, 4, 5 ], [ 2, 4, 6 ] ]
gap> Cartesian( [1,2,2], [1,1,2] );
```

```
[ [ 1, 1 ], [ 1, 1 ], [ 1, 2 ], [ 2, 1 ], [ 2, 1 ], [ 2, 2 ],
  [ 2, 1 ], [ 2, 1 ], [ 2, 2 ] ]
```

The function `Tuples` computes the k-fold cartesian product of a list.

3.22 Number

`Number(` *list* `)`
`Number(` *list*`,` *func* `)`

In the first form `Number` returns the number of bound entries in the list *list*.

For lists that contain no holes `Number`, `Length` (see 3.4), and `Size` return the same value. For lists with holes `Number` returns the number of bound entries, `Length` returns the largest index of a bound entry, and `Size` signals an error.

`Number` returns the number of elements of the list *list* for which the unary function *func* returns `true`. If an element for which *func* returns `true` appears several times in *list* it will also be counted several times. *func* must return either `true` or `false` for every element of *list*, otherwise an error is signalled.

```
gap> Number( [ 2, 3, 5, 7 ] );
4
gap> Number( [, 2, 3,, 5,, 7,,,, 11 ] );
5
gap> Number( [1..20], IsPrime );
8
gap> Number( [ 1, 3, 4, -4, 4, 7, 10, 6 ], IsPrimePowerInt );
4
gap> Number( [ 1, 3, 4, -4, 4, 7, 10, 6 ],
>            n -> IsPrimePowerInt(n) and n mod 2 <> 0 );
2
```

`Filtered` (see 3.24) allows you to extract the elements of a list that have a certain property.

3.23 Collected

`Collected(` *list* `)`

`Collected` returns a new list *new* that contains for each different element *elm* of *list* a list of two elements, the first element is *elm* itself, and the second element is the number of times *elm* appears in *list*. The order of those pairs in *new* corresponds to the ordering of the elements *elm*, so that the result is sorted.

```
gap> Factors( Factorial( 10 ) );
[ 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 5, 5, 7 ]
gap> Collected( last );
[ [ 2, 8 ], [ 3, 4 ], [ 5, 2 ], [ 7, 1 ] ]
gap> Collected( last );
[ [ [ 2, 8 ], 1 ], [ [ 3, 4 ], 1 ], [ [ 5, 2 ], 1 ], [ [ 7, 1 ], 1 ] ]
```

3.24 Filtered

`Filtered( list, func )`

`Filtered` returns a new list that contains those elements of the list *list* for which the unary function *func* returns `true`. The order of the elements in the result is the same as the order of the corresponding elements of *list*. If an element, for which *func* returns `true` appears several times in *list* it will also appear the same number of times in the result. *list* may contain holes, they are ignored by `Filtered`. *func* must return either `true` or `false` for every element of *list*, otherwise an error is signalled.

```
gap> Filtered( [1..20], IsPrime );
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
gap> Filtered( [ 1, 3, 4, -4, 4, 7, 10, 6 ], IsPrimePowerInt );
[ 3, 4, 4, 7 ]
gap> Filtered( [ 1, 3, 4, -4, 4, 7, 10, 6 ],
>              n -> IsPrimePowerInt(n) and n mod 2 <> 0 );
[ 3, 7 ]
```

The result is a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the argument list (see 3.8).

`Sublist` (see 3.20) allows you to extract elements of a list according to indices given in another list.

3.25 ForAll

`ForAll( list, func )`

`ForAll` returns `true` if the unary function *func* returns `true` for all elements of the list *list* and `false` otherwise. *list* may contain holes. *func* must return either `true` or `false` for every element of *list*, otherwise an error is signalled.

```
gap> ForAll( [1..20], IsPrime );
false
gap> ForAll( [2,3,4,5,8,9], IsPrimePowerInt );
true
gap> ForAll( [2..14], n -> IsPrimePowerInt(n) or n mod 2 = 0 );
true
```

`ForAny` (see 3.26) allows you to test if any element of a list satisfies a certain property.

3.26 ForAny

`ForAny( list, func )`

`ForAny` returns `true` if the unary function *func* returns `true` for at least one element of the list *list* and `false` otherwise. *list* may contain holes. *func* must return either `true` or `false` for every element of *list*, otherwise `ForAny` signals an error.

```
gap> ForAny( [1..20], IsPrime );
true
gap> ForAny( [2,3,4,5,8,9], IsPrimePowerInt );
true
gap> ForAny( [2..14],
>    n -> IsPrimePowerInt(n) and n mod 5 = 0 and not IsPrime(n) );
false
```

`ForAll` (see 3.25) allows you to test if all elements of a list satisfies a certain propertie.

3.27 First

`First(` *list*`,` *func* `)`

`First` returns the first element of the list *list* for which the unary function *func* returns `true`. *list* may contain holes. *func* must return either `true` or `false` for every element of *list*, otherwise an error is signalled. If *func* returns `false` for every element of *list* an error is signalled.

```
gap> First( [10^7..10^8], IsPrime );
10000019
gap> First( [10^5..10^6],
>           n -> not IsPrime(n) and IsPrimePowerInt(n) );
100489
```

`PositionProperty` (see 3.16) allows you to find the position of the first element in a list that satisfies a certain property.

3.28 Sort

`Sort(` *list* `)`
`Sort(` *list*`,` *func* `)`

`Sort` sorts the list *list* in increasing order, using shellsort. In the first form `Sort` uses the operator `<` to compare the elements. In the second form `Sort` uses the function *func* to compare elements. This function must be a function taking two arguments that returns `true` if the first is strictly smaller than the second and `false` otherwise.

`Sort` does not return anything, since it changes the argument *list*. Use `ShallowCopy` (see 5.12) if you want to keep *list*. Use `Reversed` (see 3.19) if you want to get a new list sorted in decreasing order.

It is possible to sort lists that contain multiple elements which compare equal. It is not guaranteed that those elements keep their relative order, i.e., `Sort` is not stable.

```
gap> list := [ 5, 4, 6, 1, 7, 5 ];;  Sort( list );  list;
[ 1, 4, 5, 5, 6, 7 ]
gap> list := [ [0,6], [1,2], [1,3], [1,5], [0,4], [3,4] ];;
gap> Sort( list, function(v,w) return v*v < w*w; end ); list;
[ [ 1, 2 ], [ 1, 3 ], [ 0, 4 ], [ 3, 4 ], [ 1, 5 ], [ 0, 6 ] ]
```

```
# sorted according to the Euclidian distance from [0,0]
gap> list := [ [0,6], [1,3], [3,4], [1,5], [1,2], [0,4], ];;
gap> Sort( list, function(v,w) return v[1] < w[1]; end ); list;
[ [ 0, 6 ], [ 0, 4 ], [ 1, 3 ], [ 1, 5 ], [ 1, 2 ], [ 3, 4 ] ]
# note the random order of the elements with equal first component
```

`SortParallel` (see 3.29) allows you to sort a list and apply the exchanges that are necessary to another list in parallel. `Sortex` (see 3.30) sorts a list and returns the sorting permutation.

3.29 SortParallel

`SortParallel(` *list1*`,` *list2* `)`
`SortParallel(` *list1*`,` *list2*`,` *func* `)`

`SortParallel` sorts the list *list1* in increasing order just as `Sort` (see 3.28) does. In parallel it applies the same exchanges that are necessary to sort *list1* to the list *list2*, which must of course have at least as many elements as *list1* does.

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := [ 2, 3, 5, 7, 8, 9 ];;
gap> SortParallel( list1, list2 );
gap> list1;
[ 1, 4, 5, 5, 6, 7 ]
gap> list2;
[ 7, 3, 2, 9, 5, 8 ]    # [ 7, 3, 9, 2, 5, 8 ] is also possible
```

`Sortex` (see 3.30) sorts a list and returns the sorting permutation.

3.30 Sortex

`Sortex(` *list* `)`

`Sortex` sorts the list *list* and returns the permutation that must be applied to *list* to obtain the sorted list.

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := Copy( list1 );;
gap> perm := Sortex( list1 );
(1,3,5,6,4)
gap> list1;
[ 1, 4, 5, 5, 6, 7 ]
gap> Permuted( list2, perm );
[ 1, 4, 5, 5, 6, 7 ]
```

`Permuted` (see 3.31) allows you to rearrange a list according to a given permutation.

3.31 Permuted

`Permuted(` *list*`,` *perm* `)`

`Permuted` returns a new list *new* that contains the elements of the list *list* permuted according to the permutation *perm*. That is *new*`[`*i*`^`*perm*`] = `*list*`[`*i*`]`.

```
gap> Permuted( [ 5, 4, 6, 1, 7, 5 ], (1,3,5,6,4) );
[ 1, 4, 5, 5, 6, 7 ]
```

`Sortex` (see 3.30) allows you to compute the permutation that must be applied to a list to get the sorted list.

3.32 Product

`Product(` *list* `)`
`Product(` *list*`,` *func* `)`

In the first form `Product` returns the product of the elements of the list *list*, which must have no holes. If *list* is empty, the integer 1 is returned.

In the second form `Product` applies the function *func* to each element of the list *list*, which must have no holes, and multiplies the results. If the *list* is empty, the integer 1 is returned.

```
gap> Product( [ 2, 3, 5, 7, 11, 13, 17, 19 ] );
9699690
gap> Product( [1..10], x->x^2 );
13168189440000
gap> Product( [ (1,2), (1,3), (1,4), (2,3), (2,4), (3,4) ] );
(1,4)(2,3)
```

`Sum` (see 3.33) computes the sum of the elements of a list.

3.33 Sum

`Sum(` *list* `)`
`Sum(` *list*`,` *func* `)`

In the first form `Sum` returns the sum of the elements of the list *list*, which must have no holes. If *list* is empty 0 is returned.

In the second form `Sum` applies the function *func* to each element of the list *list*, which must have no holes, and sums the results. If the *list* is empty 0 is returned.

```
gap> Sum( [ 2, 3, 5, 7, 11, 13, 17, 19 ] );
77
gap> Sum( [1..10], x->x^2 );
385
gap> Sum( [ [1,2], [3,4], [5,6] ] );
[ 9, 12 ]
```

`Product` (see 3.32) computes the product of the elements of a list.

3.34 Maximum

`Maximum(` *obj1*`,` *obj2*`..` `)`
`Maximum(` *list* `)`

`Maximum` returns the maximum of its arguments, i.e., that argument obj_i for which $obj_k <= obj_i$ for all k. In its second form `Maximum` takes a list *list* and returns the maximum of the elements of this list.

Typically the arguments or elements of the list respectively will be integers, but actually they can be objects of an arbitrary type. This works because any two objects can be compared using the `<` operator.

```
gap> Maximum( -123, 700, 123, 0, -1000 );
700
gap> Maximum( [ -123, 700, 123, 0, -1000 ] );
700
gap> Maximum( [ 1, 2 ], [ 0, 15 ], [ 1, 5 ], [ 2, -11 ] );
[ 2, -11 ]          # lists are compared elementwise
```

3.35 Minimum

`Minimum(` *obj1*`,` *obj2*`..` `)`
`Minimum(` *list* `)`

`Minimum` returns the minimum of its arguments, i.e., that argument obj_i for which $obj_i <= obj_k$ for all k. In its second form `Minimum` takes a list *list* and returns the minimum of the elements of this list.

Typically the arguments or elements of the list respectively will be integers, but actually they can be objects of an arbitrary type. This works because any two objects can be compared using the `<` operator.

```
gap> Minimum( -123, 700, 123, 0, -1000 );
-1000
gap> Minimum( [ -123, 700, 123, 0, -1000 ] );
-1000
gap> Minimum( [ 1, 2 ], [ 0, 15 ], [ 1, 5 ], [ 2, -11 ] );
[ 0, 15 ]          # lists are compared elementwise
```

3.36 Iterated

`Iterated(` *list*`,` *f* `)`

`Iterated` returns the result of the iterated application of the function f, which must take two arguments, to the elements of *list*. More precisely `Iterated` returns the result of the following application, *f*`(..`*f*`(` *f*`(` *list*`[1],` *list*`[2] ),` *list*`[3] ),..,`*list*`[`n`] )`.

```
gap> Iterated( [ 126, 66, 105 ], Gcd );
3
```

3.37 RandomList

`RandomList( list )`

`RandomList` returns a random element of the list *list*. The results are equally distributed, i.e., all elements are equally likely to be selected.

```
gap> RandomList( [1..200] );
192
gap> RandomList( [1..200] );
152
gap> RandomList( [ [ 1, 2 ], 3, [ 4, 5 ], 6 ] );
[ 4, 5 ]
```

`RandomSeed( n )`

`RandomSeed` seeds the pseudo random number generator `RandomList`. Thus to reproduce a computation exactly you can call `RandomSeed` each time before you start the computation. When GAP is started the pseudo random number generator is seeded with 1.

```
gap> RandomSeed(1);  RandomList([1..100]);  RandomList([1..100]);
96
76
gap> RandomSeed(1);  RandomList([1..100]);  RandomList([1..100]);
96
76
```

`RandomList` is called by all random functions for domains.

Chapter 4

Sets

A very important mathematical concept, maybe the most important of all, are sets. Mathematically a **set** is an abstract object such that each object is either an element of the set or it is not. So a set is a collection like a list, and in fact GAP uses lists to represent sets. Note that this of course implies that GAP only deals with finite sets.

Unlike a list a set must not contain an element several times. It simply makes no sense to say that an object is twice an element of a set, because an object is either an element of a set, or it is not. Therefore the list that is used to represent a set has no duplicates, that is, no two elements of such a list are equal.

Also unlike a list a set does not impose any ordering on the elements. Again it simply makes no sense to say that an object is the first or second etc. element of a set, because, again, an object is either an element of a set, or it is not. Since ordering is not defined for a set we can put the elements in any order into the list used to represent the set. We put the elements sorted into the list, because this ordering is very practical. For example if we convert a list into a set we have to remove duplicates, which is very easy to do after we have sorted the list, since then equal elements will be next to each other.

In short sets are represented by sorted lists without holes and duplicates in GAP. Such a list is in this document called a proper set. Note that we guarantee this representation, so you may make use of the fact that a set is represented by a sorted list in your functions.

In some contexts, we also want to talk about multisets. A **multiset** is like a set, except that an element may appear several times in a multiset. Such multisets are represented by sorted lists with holes that may have duplicates.

The first section in this chapter describes the functions to test if an object is a set and to convert objects to sets (see 4.1 and 4.2).

The next section describes the function that tests if two sets are equal (see 4.3).

The next sections describe the destructive functions that compute the standard set operations for sets (see 4.4, 4.5, 4.6, 4.7, and 4.8).

The last section tells you more about sets and their internal representation (see 4.10).

All set theoretic functions, especially `Intersection` and `Union`, also accept sets as arguments. Thus all functions described in the chapter Domains in the GAP-manual are applicable to sets

(see 4.9).

Since sets are just a special case of lists, all the operations and functions for lists, especially the membership test (see 3.13), can be used for sets just as well.

4.1 IsSet

`IsSet(` *obj* `)`

`IsSet` returns `true` if the object *obj* is a set and `false` otherwise. An object is a set if it is a sorted lists without holes or duplicates. Will cause an error if evaluation of *obj* is an unbound variable.

```
gap> IsSet( [] );
true
gap> IsSet( [ 2, 3, 5, 7, 11 ] );
true
gap> IsSet( [, 2, 3,, 5,, 7,,,, 11 ] );
false        # this list contains holes
gap> IsSet( [ 11, 7, 5, 3, 2 ] );
false        # this list is not sorted
gap> IsSet( [ 2, 2, 3, 5, 5, 7, 11, 11 ] );
false        # this list contains duplicates
gap> IsSet( 235711 );
false        # this argument is not even a list
```

4.2 Set

`Set(` *list* `)`

`Set` returns a new proper set, which is represented as a sorted list without holes or duplicates, containing the elements of the list *list*.

`Set` returns a new list even if the list *list* is already a proper set, in this case it is equivalent to `ShallowCopy` (see 5.12). Thus the result is a new list that is not identical to any other list. The elements of the result are however identical to elements of *list*. If *list* contains equal elements, it is not specified to which of those the element of the result is identical (see 3.8).

```
gap> Set( [3,2,11,7,2,,5] );
[ 2, 3, 5, 7, 11 ]
gap> Set( [] );
[ ]
```

4.3 SetIsEqual

`SetIsEqual(` *list1* `,` *list2* `)`

`SetIsEqual` returns `true` if the two lists *list1* and *list2* are equal **when viewed as sets**, and

`false` otherwise. *list1* and *list2* are equal if every element of *list1* is also an element of *list2* and if every element of *list2* is also an element of *list1*.

If both lists are proper sets then they are of course equal if and only if they are also equal as lists. Thus `SetIsEqual( list1, list2 )` is equivalent to `Set( list1 ) = Set( list2 )` (see 4.2), but the former is more efficient.

```
gap> SetIsEqual( [2,3,5,7,11], [11,7,5,3,2] );
true
gap> SetIsEqual( [2,3,5,7,11], [2,3,5,7,11,13] );
false
```

4.4 SetAdd

`SetAdd( set, elm )`

`SetAdd` adds *elm*, which may be an elment of an arbitrary type, to the set *set*, which must be a proper set, otherwise an error will be signalled. If *elm* is already an element of the set *set*, the set is not changed. Otherwise *elm* is inserted at the correct position such that *set* is again a set afterwards.

```
gap> s := [2,3,7,11];;
gap> SetAdd( s, 5 );  s;
[ 2, 3, 5, 7, 11 ]
gap> SetAdd( s, 13 );  s;
[ 2, 3, 5, 7, 11, 13 ]
gap> SetAdd( s, 3 );  s;
[ 2, 3, 5, 7, 11, 13 ]
```

`SetRemove` (see 4.5) is the counterpart of `SetAdd`.

4.5 SetRemove

`SetRemove( set, elm )`

`SetRemove` removes the element *elm*, which may be an object of arbitrary type, from the set *set*, which must be a set, otherwise an error will be signalled. If *elm* is not an element of *set* nothing happens. If *elm* is an element it is removed and all the following elements in the list are moved one position forward.

```
gap> s := [ 2, 3, 4, 5, 6, 7 ];;
gap> SetRemove( s, 6 );
gap> s;
[ 2, 3, 4, 5, 7 ]
gap> SetRemove( s, 10 );
gap> s;
[ 2, 3, 4, 5, 7 ]
```

`SetAdd` (see 4.4) is the counterpart of `SetRemove`.

4.6 SetUnite

`SetUnite( set1, set2 )`

`SetUnite` unites the set *set1* with the set *set2*. This is equivalent to adding all the elements in *set2* to *set1* (see 4.4). *set1* must be a proper set, otherwise an error is signalled. *set2* may also be list that is not a proper set, in which case `SetUnite` silently applies `Set` to it first (see 4.2). `SetUnite` returns nothing, it is only called to change *set1*.

```
gap> set := [ 2, 3, 5, 7, 11 ];;
gap> SetUnite( set, [ 4, 8, 9 ] );  set;
[ 2, 3, 4, 5, 7, 8, 9, 11 ]
gap> SetUnite( set, [ 16, 9, 25, 13, 16 ] );  set;
[ 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 25 ]
```

The function `UnionSet` (see 4.9) is the nondestructive counterpart to the destructive procedure `SetUnite`.

4.7 SetIntersect

`SetIntersect( set1, set2 )`

`SetIntersect` intersects the set *set1* with the set *set2*. This is equivalent to removing all the elements that are not in *set2* from *set1* (see 4.5). *set1* must be a set, otherwise an error is signalled. *set2* may be a list that is not a proper set, in which case `SetIntersect` silently applies `Set` to it first (see 4.2). `SetIntersect` returns nothing, it is only called to change *set1*.

```
gap> set := [ 2, 3, 4, 5, 7, 8, 9, 11, 13, 16 ];;
gap> SetIntersect( set, [ 3, 5, 7, 9, 11, 13, 15, 17 ] );  set;
[ 3, 5, 7, 9, 11, 13 ]
gap> SetIntersect( set, [ 9, 4, 6, 8 ] );  set;
[ 9 ]
```

The function `IntersectionSet` (see 4.9) is the nondestructive counterpart to the destructive procedure `SetIntersect`.

4.8 SetSubtract

`SetSubtract( set1, set2 )`

`SetSubtract` subtracts the set *set2* from the set *set1*. This is equivalent to removing all the elements in *set2* from *set1* (see 4.5). *set1* must be a proper set, otherwise an error is signalled. *set2* may be a list that is not a proper set, in which case `SetSubtract` applies `Set` to it first (see 4.2). `SetSubtract` returns nothing, it is only called to change *set1*.

```
gap> set := [ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ];;
gap> SetSubtract( set, [ 6, 10 ] );  set;
[ 2, 3, 4, 5, 7, 8, 9, 11 ]
gap> SetSubtract( set, [ 9, 4, 6, 8 ] );  set;
```

```
[ 2, 3, 5, 7, 11 ]
```

The function `Difference` is the nondestructive counterpart to destructive the procedure `SetSubtract`.

4.9 Set Functions for Sets

As was already mentioned in the introduction to this chapter all domain functions also accept sets as arguments. Thus all functions described in the chapter Domains in the GAP-manual are applicable to sets. This section describes those functions where it might be helpful to know the implementation of those functions for sets.

`IsSubset(` *set1*`,` *set2* `)`

This is implemented by `SetIsSubset`, which you can call directly to save a little bit of time. Either argument to `SetIsSubset` may also be a list that is not a proper set, in which case `IsSubset` silently applies `Set` (see 4.2) to it first.

`Union(` *set1*`,` *set2* `)`

This is implemented by `UnionSet`, which you can call directly to save a little bit of time. Note that `UnionSet` only accepts two sets, unlike `Union`, which accepts several sets or a list of sets. The result of `UnionSet` is a new set, represented as a sorted list without holes or duplicates. Each argument to `UnionSet` may also be a list that is not a proper set, in which case `UnionSet` silently applies `Set` (see 4.2) to this argument. `UnionSet` is implemented in terms of its destructive counterpart `SetUnite` (see 4.6).

`Intersection(` *set1*`,` *set2* `)`

This is implemented by `IntersectionSet`, which you can call directly to save a little bit of time. Note that `IntersectionSet` only accepts two sets, unlike `Intersection`, which accepts several sets or a list of sets. The result of `IntersectionSet` is a new set, represented as a sorted list without holes or duplicates. Each argument to `IntersectionSet` may also be a list that is not a proper set, in which case `IntersectionSet` silently applies `Set` (see 4.2) to this argument. `IntersectionSet` is implemented in terms of its destructive counterpart `SetIntersect` (see 4.7).

The result of `IntersectionSet` and `UnionSet` is always a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of *set1*. If *set1* is not a proper list it is not specified to which of a number of equal elements in *set1* the element in the result is identical (see 3.8).

4.10 More about Sets

In the previous section we defined a proper set as a sorted list without holes or duplicates. This representation is not only nice to use, it is also a good internal representation supporting efficient algorithms. For example the `in` operator can use binary instead of a linear search since a set is sorted. For another example `Union` only has to merge the sets.

However, all those set functions also allow lists that are not proper sets, silently making a

copy of it and converting this copy to a set. Suppose all the functions would have to test their arguments every time, comparing each element with its successor, to see if they are proper sets. This would chew up most of the performance advantage again. For example suppose `in` would have to run over the whole list, to see if it is a proper set, so it could use the binary search. That would be ridiculous.

To avoid this a list that is a proper set may, but need not, have an internal flag set that tells those functions that this list is indeed a proper set. Those functions do not have to check this argument then, and can use the more efficient algorithms. This section tells you when a proper set obtains this flag, so you can write your functions in such a way that you make best use of the algorithms.

The results of `Set`, `Difference`, `Intersection` and `Union` are known to be sets by construction, and thus have the flag set upon creation.

If an argument to `IsSet`, `SetIsEqual`, `IsSubset`, `Set`, `Difference`, `Intersection` or `Union` is a proper set, that does not yet have the flag set, those functions will notice that and set the flag for this set. Note that `in` will use linear search if the right operand does not have the flag set, will therefore not detect if it is a proper set and will, unlike the functions above, never set the flag.

If you change a proper set, that does have this flag set, by assignment, `Add` or `Append` the set will generally lose it flag, even if the change is such that the resulting list is still a proper set. However if the set has more than 100 elements and the value assigned or added is not a list and not a record and the resulting list is still a proper set than it will keep the flag. Note that changing a list that is not a proper set will never set the flag, even if the resulting list is a proper set. Such a set will obtain the flag only if it is passed to a set function.

Suppose you have built a proper set in such a way that it does not have the flag set, and that you now want to perform lots of membership tests. Then you should call `IsSet` with that set as an argument. If it is indeed a proper set `IsSet` will set the flag, and the subsequent `in` operations will use the more efficient binary search. You can think of the call to `IsSet` as a hint to GAP that this list is a proper set.

There is no way you can set the flag for an ordinary list without going through the checking in `IsSet`. The internal functions depend so much on the fact that a list with this flag set is indeed sorted and without holes and duplicates that the risk would be too high to allow setting the flag without such a check.

Chapter 5

Records

Records are next to lists the most important way to collect objects together. A record is a collection of **components**. Each component has a unique **name**, which is an identifier that distinguishes this component, and a **value**, which is an object of arbitrary type. We often abbreviate **value of a component** to **element**. We also say that a record **contains** its elements. You can access and change the elements of a record using its name.

Record literals are written by writing down the components in order between `rec(` and `)`, and separating them by commas `,`. Each component consists of the name, the assignment operator `:=`, and the value. The **empty record**, i.e., the record with no components, is written as `rec()`.

```
gap> rec( a := 1, b := "2" );      #  a record with two components
rec(
  a := 1,
  b := "2" )
gap> rec( a := 1, b := rec( c := 2 ) );      #  record may contain records
rec(
  a := 1,
  b := rec(
      c := 2 ) )
```

Records usually contain elements of various types, i.e., they are usually not homogeneous like lists.

The first section in this chapter tells you how you can access the elements of a record (see 5.1).

The next sections tell you how you can change the elements of a record (see 5.2 and 5.3).

The next sections describe the operations that are available for records (see 5.4, 5.5, 5.6, and 5.7).

The next section describes the function that tests if an object is a record (see 5.8).

The next sections describe the functions that test whether a record has a component with a given name, and delete such a component (see 5.9 and 5.10). Those functions are also applicable to lists.

The final sections describe the functions that create a copy of a record (see 5.11 and 5.12). Again those functions are also applicable to lists.

5.1 Accessing Record Elements

rec`.`*name*

The above construct evaluates to the value of the record component with the name *name* in the record *rec*. Note that the *name* is not evaluated, i.e., it is taken literal.

```
gap> r := rec( a := 1, b := 2 );;
gap> r.a;
1
gap> r.b;
2
```

rec`.(`*name*`)`

This construct is similar to the above construct. The difference is that the second operand *name* is evaluated. It must evaluate to a string or an integer otherwise an error is signalled. The construct then evaluates to the element of the record *rec* whose name is, as a string, equal to *name*.

```
gap> old := rec( a := 1, b := 2 );;
gap> new := rec();
rec(
   )
gap> for i  in RecFields( old )  do
>        new.(i) := old.(i);
>    od;
gap> new;
rec(
  a := 1,
  b := 2 )
```

If *rec* does not evaluate to a record, or if *name* does not evaluate to a string, or if *rec*`.`*name* is unbound, an error is signalled. As usual you can leave the break loop with `quit;`. On the other hand you can return a result to be used in place of the record element by `return` *expr*`;`.

5.2 Record Assignment

rec`.`*name* `:=` *obj*`;`

The record assignment assigns the object *obj*, which may be an object of arbitrary type, to the record component with the name *name*, which must be an identifier, of the record *rec*. That means that accessing the element with name *name* of the record *rec* will return *obj* after this assignment. If the record *rec* has no component with the name *name*, the record is automatically extended to make room for the new component.

```
gap> r := rec( a := 1, b := 2 );;
gap> r.a := 10;;  r;
rec(
  a := 10,
  b := 2 )
gap> r.c := 3;;  r;
rec(
  a := 10,
  b := 2,
  c := 3 )
```

The function `IsBound` (see 5.9) can be used to test if a record has a component with a certain name, the function `Unbind` (see 5.10) can be used to remove a component with a certain name again.

Note that assigning to a record changes the record. The ability to change an object is only available for lists and records (see 5.3).

rec`.(`*name*`)` `:=` *obj*`;`

This construct is similar to the above construct. The difference is that the second operand *name* is evaluated. It must evaluate to a string or an integer otherwise an error is signalled. The construct then assigns *obj* to the record component of the record *rec* whose name is, as a string, equal to *name*.

If *rec* does not evaluate to a record, *name* does not evaluate to a string, or *obj* is a call to a function that does not return a value, e.g., `Print`, an error is signalled. As usual you can leave the break loop with `quit;`. On the other hand you can continue the assignment by returning a record in the first case, a string in the second, or an object to be assigned in the third, using `return` *expr*`;`.

5.3 Identical Records

With the record assignment (see 5.2) it is possible to change a record. The ability to change an object is only available for lists and records. This section describes the semantic consequences of this fact.

You may think that in the following example the second assignment changes the integer, and that therefore the above sentence, which claimed that only records and lists can be changed, is wrong.

```
i := 3;
i :=  i + 1;
```

But in this example not the **integer** `3` is changed by adding one to it. Instead the **variable** `i` is changed by assigning the value of `i+1`, which happens to be `4`, to `i`. The same thing happens in the following example

```
r := rec( a := 1 );
r := rec( a := 1, b := 2 );
```

The second assignment does not change the first record, instead it assigns a new record to the

variable `r`. On the other hand, in the following example the record is changed by the second assignment.

```
r := rec( a := 1 );
r.b := 2;
```

To understand the difference first think of a variable as a name for an object. The important point is that a record can have several names at the same time. An assignment *var* `:=` *record*`;` means in this interpretation that *var* is a name for the object *record*. At the end of the following example `r2` still has the value `rec( a := 1 )` as this record has not been changed and nothing else has been assigned to `r2`.

```
r1 := rec( a := 1 );
r2 := r1;
r1 := rec( a := 1, b := 2 );
```

But after the following example the record for which `r2` is a name has been changed and thus the value of `r2` is now `rec( a := 1, b := 2 )`.

```
r1 := rec( a := 1 );
r2 := r1;
r1.b := 2;
```

We shall say that two records are **identical** if changing one of them by a record assignment also changes the other one. This is slightly incorrect, because if **two** records are identical, there are actually only two names for **one** record. However, the correct usage would be very awkward and would only add to the confusion. Note that two identical records must be equal, because there is only one records with two different names. Thus identity is an equivalence relation that is a refinement of equality.

Let us now consider under which circumstances two records are identical.

If you enter a record literal then the record denoted by this literal is a new record that is not identical to any other record. Thus in the following example `r1` and `r2` are not identical, though they are equal of course.

```
r1 := rec( a := 1 );
r2 := rec( a := 1 );
```

Also in the following example, no records in the list `l` are identical.

```
l := [];
for i  in [1..10]  do
    l[i] := rec( a := 1 );
od;
```

If you assign a record to a variable no new record is created. Thus the record value of the variable on the left hand side and the record on the right hand side of the assignment are identical. So in the following example `r1` and `r2` are identical records.

```
r1 := rec( a := 1 );
r2 := r1;
```

If you pass a record as argument, the old record and the argument of the function are identical. Also if you return a record from a function, the old record and the value of the function call are identical. So in the following example `r1` and `r2` are identical record

```
r1 := rec( a := 1 );
f := function ( r )  return r;  end;
r2 := f( r1 );
```

The functions `Copy` and `ShallowCopy` (see 5.11 and 5.12) accept a record and return a new record that is equal to the old record but that is **not** identical to the old record. The difference between `Copy` and `ShallowCopy` is that in the case of `ShallowCopy` the corresponding elements of the new and the old records will be identical, whereas in the case of `Copy` they will only be equal. So in the following example `r1` and `r2` are not identical records.

```
r1 := rec( a := 1 );
r2 := Copy( r1 );
```

If you change a record it keeps its identity. Thus if two records are identical and you change one of them, you also change the other, and they are still identical afterwards. On the other hand, two records that are not identical will never become identical if you change one of them. So in the following example both `r1` and `r2` are changed, and are still identical.

```
r1 := rec( a := 1 );
r2 := r1;
r1.b := 2;
```

5.4 Comparisons of Records

rec1 `=` *rec2*
rec1 `<>` *rec2*

The equality operator `=` returns `true` if the record *rec1* is equal to the record *rec2* and `false` otherwise. The inequality operator `<>` returns `true` if the record *rec1* is not equal to *rec2* and `false` otherwise.

Usually two records are considered equal, if for each component of one record the other record has a component of the same name with an equal value and vice versa. You can also compare records with other objects, they are of course different, unless the record has a special comparison function (see below) that says otherwise.

```
gap> rec( a := 1, b := 2 ) = rec( b := 2, a := 1 );
true
gap> rec( a := 1, b := 2 ) = rec( a := 2, b := 1 );
false
gap> rec( a := 1 ) = rec( a := 1, b := 2 );
false
gap> rec( a := 1 ) = 1;
false
```

However a record may contain a special `operations` record that contains a function that is called when this record is an operand of a comparison. The precise mechanism is as follows. If the operand of the equality operator `=` is a record, and if this record has an element with the name `operations` that is a record, and if this record has an element with the name `=` that is a function, then this function is called with the operands of `=` as arguments, and the value of the operation is the result returned by this function. In this case a record may also

be equal to an object of another type if this function says so. It is probably not a good idea to define a comparison function in such a way that the resulting relation is not an equivalence relation, i.e., not reflexive, symmetric, and transitive. Note that there is no corresponding `<>` function, because *left* `<>` *right* is implemented as `not` *left* `=` *right*.

The following example shows one piece of the definition of residue classes, using record operations. Of course this is far from a complete implementation. Note that the `=` must be quoted, so that it is taken as an identifier (see 2.5).

```
gap> ResidueClassOps := rec( );;
gap> ResidueClassOps.\= := function ( l, r )
>   return (l.modulus = r.modulus)
>     and (l.representative-r.representative) mod l.modulus = 0;
> end;;
gap> ResidueClass := function ( representative, modulus )
>   return rec(
>     representative := representative,
>     modulus        := modulus,
>     operations     := ResidueClassOps );
> end;;
gap> l := ResidueClass( 13, 23 );;
gap> r := ResidueClass( -10, 23 );;
gap> l = r;
true
gap> l = ResidueClass( 10, 23 );
false
```

rec1 `<` *rec2*
rec1 `<=` *rec2*
rec1 `>` *rec2*
rec1 `>=` *rec2*

The operators `<`, `<=`, `>`, and `>=` evaluate to `true` if the record *rec1* is less than, less than or equal to, greater than, and greater than or equal to the record *rec2*, and to `false` otherwise.

To compare records we imagine that the components of both records are sorted according to their names. Then the records are compared lexicographically with unbound elements considered smaller than anything else. Precisely one record *rec1* is considered less than another record *rec2* if *rec2* has a component with name *name2* and either *rec1* has no component with this name or *rec1*`.`*name2* `<` *rec2*`.`*name2* and for each component of *rec1* with name *name1* `<` *name2* *rec2* has a component with this name and *rec1*`.`*name1* `=` *rec2*`.`*name1*. Records may also be compared with objects of other types, they are greater than anything else, unless the record has a special comparison function (see below) that says otherwise.

```
gap> rec( a := 1, b := 2 ) < rec( b := 2, a := 1 );
false    # they are equal
gap> rec( a := 1, b := 2 ) < rec( a := 2, b := 0 );
true    # the a elements are compared first and 1 is less than 2
gap> rec( a := 1 ) < rec( a := 1, b := 2 );
true    # unbound is less than 2
```

```
gap> rec( a := 1 ) < rec( a := 0, b := 2 );
false     # the a elements are compared first and 0 is less than 1
gap> rec( b := 1 ) < rec( b := 0, a := 2 );
true     # the a-s are compared first and unbound is less than 2
gap> rec( a := 1 ) < 1;
false     # other objects are less than records
```

However a record may contain a special `operations` record that contains a function that is called when this record is an operand of a comparison. The precise mechanism is as follows. If the operand of the equality operator `<` is a record, and if this record has an element with the name `operations` that is a record, and if this record has an element with the name `<` that is a function, then this function is called with the operands of `<` as arguments, and the value of the operation is the result returned by this function. In this case a record may also be smaller than an object of another type if this function says so. It is probably not a good idea to define a comparison function in such a way that the resulting relation is not an ordering relation, i.e., not antisymmetric, and transitive. Note that there are no corresponding `<=`, `>`, and `>=` functions, since those operations are implemented as `not` *right* `<`*left*, *right* `<`*left*, and `not` *left* `<`*right* respectively.

The following example shows one piece of the definition of residue classes, using record operations. Of course this is far from a complete implementation. Note that the `<` must be quoted, so that it is taken as an identifier (see 2.5).

```
gap> ResidueClassOps := rec( );;
gap> ResidueClassOps.\< := function ( l, r )
>   if l.modulus <> r.modulus  then
>     Error("<l> and <r> must have the same modulus");
>   fi;
>   return   l.representative mod l.modulus
>          < r.representative mod r.modulus;
> end;;
gap> ResidueClass := function ( representative, modulus )
>   return rec(
>     representative := representative,
>     modulus        := modulus,
>     operations     := ResidueClassOps );
> end;;
gap> l := ResidueClass( 13, 23 );;
gap> r := ResidueClass( -1, 23 );;
gap> l < r;
true     # 13 is less than 22
gap> l < ResidueClass( 10, 23 );
false     # 10 is less than 13
```

5.5 Operations for Records

Usually no operations are defined for record. However a record may contain a special `operations` record that contains functions that are called when this record is an operand

of a binary operation. This mechanism is detailed below for the addition.

obj `+` *rec*, *rec* `+` *obj*

If either operand is a record, and if this record contains an element with name `operations` that is a record, and if this record in turn contains an element with the name `+` that is a function, then this function is called with the two operands as arguments, and the value of the addition is the value returned by that function. If both operands are records with such a function *rec*`.operations.+`, then the function of the **right** operand is called. If either operand is a record, but neither operand has such a function *rec*`.operations.+`, an error is signalled.

obj `-` *rec*, *rec* `-` *obj*
obj `*` *rec*, *rec* `*` *obj*
obj `/` *rec*, *rec* `/` *obj*
obj `mod` *rec*, *rec* `mod` *obj*
obj `^` *rec*, *rec* `^` *obj*

This is evaluated similar, but the functions must obviously be called `-`, `*`, `/`, `mod`, `^` respectively.

The following example shows one piece of the definition of a residue classes, using record operations. Of course this is far from a complete implementation. Note that the `*` must be quoted, so that it is taken as an identifier (see 2.5).

```
gap> ResidueClassOps := rec( );;
gap> ResidueClassOps.\* := function ( l, r )
>   if l.modulus <> r.modulus  then
>     Error("<l> and <r> must have the same modulus");
>   fi;
>   return rec(
>     representative := (l.representative * r.representative)
>                       mod l.modulus,
>     modulus        := l.modulus,
>     operations     := ResidueClassOps );
> end;;
gap> ResidueClass := function ( representative, modulus )
>   return rec(
>     representative := representative,
>     modulus        := modulus,
>     operations     := ResidueClassOps );
> end;;
gap> l := ResidueClass( 13, 23 );;
gap> r := ResidueClass( -1, 23 );;
gap> s := l * r;
rec(
  representative := 10,
  modulus := 23,
  operations := rec(
      * := function ( l, r ) ... end ) )
```

5.6 In for Records

element `in` *rec*

Usually the membership test is only defined for lists. However a record may contain a special `operations` record, that contains a function that is called when this record is the right operand of the `in` operator. The precise mechanism is as follows.

If the right operand of the `in` operator is a record, and if this record contains an element with the name `operations` that is a record, and if this record in turn contains an element with the name `in` that is a function, then this function is called with the two operands as arguments, and the value of the membership test is the value returned by that function. The function should of course return `true` or `false`.

The following example shows one piece of the definition of residue classes, using record operations. Of course this is far from a complete implementation. Note that the `in` must be quoted, so that it is taken as an identifier (see 2.5).

```
gap> ResidueClassOps := rec( );;
gap> ResidueClassOps.\in := function ( l, r )
>   if IsInt( l )  then
>     return (l - r.representative) mod r.modulus = 0;
>   else
>     return false;
>   fi;
> end;;
gap> ResidueClass := function ( representative, modulus )
>   return rec(
>     representative := representative,
>     modulus        := modulus,
>     operations     := ResidueClassOps );
> end;;
gap> l := ResidueClass( 13, 23 );;
gap> -10 in l;
true
gap> 10 in l;
false
```

5.7 Printing of Records

`Print(` *rec* `)`

If a record is printed by `Print` or by the main loop, it is usually printed as record literal, i.e., as a sequence of components, each in the format *name* `:=` *value*, separated by commas and enclosed in `rec(` and `)`.

```
gap> r := rec();;  r.a := 1;;  r.b := 2;;
gap> r;
rec(
```

```
    a := 1,
    b := 2 )
```

But if the record has an element with the name `operations` that is a record, and if this record has an element with the name `Print` that is a function, then this function is called with the record as argument. This function must print whatever the printed representation of the record should look like.

The following example shows one piece of the definition of residue classes, using record operations. Of course this is far from a complete implementation. Note that it is typical for records that mimic group elements to print as a function call that, when evaluated, will create this group element record.

```
gap> ResidueClassOps := rec( );;
gap> ResidueClassOps.Print := function ( r )
>   Print( "ResidueClass( ",
>               r.representative mod r.modulus, ", ",
>               r.modulus, " )" );
> end;;
gap> ResidueClass := function ( representative, modulus )
>   return rec(
>     representative := representative,
>     modulus        := modulus,
>     operations     := ResidueClassOps );
> end;;
gap> l := ResidueClass( 33, 23 );
ResidueClass( 10, 23 )
```

5.8 IsRec

`IsRec(` *obj* `)`

`IsRec` returns `true` if the object *obj*, which may be an object of arbitrary type, is a record, and `false` otherwise. Will signal an error if *obj* is a variable with no assigned value.

```
gap> IsRec( rec( a := 1, b := 2 ) );
true
gap> IsRec( IsRec );
false
```

5.9 IsBound

`IsBound(` *rec*`.`*name* `)`
`IsBound(` *list*`[`*n*`]` `)`

In the first form `IsBound` returns `true` if the record *rec* has a component with the name *name*, which must be an ident and `false` otherwise. *rec* must evaluate to a record, otherwise an error is signalled.

In the second form `IsBound` returns `true` if the list *list* has a element at the position n, and `false` otherwise. *list* must evaluate to a list, otherwise an error is signalled.

```
gap> r := rec( a := 1, b := 2 );;
gap> IsBound( r.a );
true
gap> IsBound( r.c );
false
gap> l := [  , 2, 3,  , 5,  , 7,  ,  ,   , 11 ];;
gap> IsBound( l[7] );
true
gap> IsBound( l[4] );
false
gap> IsBound( l[101] );
false
```

Note that `IsBound` is special in that it does not evaluate its argument, otherwise it would always signal an error when it is supposed to return `false`.

5.10 Unbind

`Unbind(` *rec*`.`*name* `)`
`Unbind(` *list*`[`n`]` `)`

In the first form `Unbind` deletes the component with the name *name* in the record *rec*. That is, after execution of `Unbind`, *rec* no longer has a record component with this name. Note that it is not an error to unbind a nonexisting record component. *rec* must evaluate to a record, otherwise an error is signalled.

In the second form `Unbind` deletes the element at the position n in the list *list*. That is, after execution of `Unbind`, *list* no longer has an assigned value at the position n. Note that it is not an error to unbind a nonexisting list element. *list* must evaluate to a list, otherwise an error is signalled.

```
gap> r := rec( a := 1, b := 2 );;
gap> Unbind( r.a );  r;
rec(
  b := 2 )
gap> Unbind( r.c );  r;
rec(
  b := 2 )
gap> l := [  , 2, 3,  5,  , 7,  ,  ,  , 11 ];;
gap> Unbind( l[3] );  l;
[ , 2,, 5,, 7,,,, 11 ]
gap> Unbind( l[4] );  l;
[ , 2,,,, 7,,,, 11 ]
```

Note that `Unbind` does not evaluate its argument, otherwise there would be no way for `Unbind` to tell which component to remove in which record, because it would only receive the value

of this component.

5.11 Copy

`Copy( obj )`

`Copy` returns a copy *new* of the object *obj*. You may apply `Copy` to objects of any type, but for objects that are not lists or records `Copy` simply returns the object itself.

For lists and records the result is a **new** list or record that is **not identical** to any other list or record (see 3.8 and 5.3). This means that you may modify this copy *new* by assignments (see 3.5 and 5.2) or by adding elements to it (see 3.6 and 3.7), without modifying the original object *obj*.

```
gap> list1 := [ 1, 2, 3 ];;
gap> list2 := Copy( list1 );
[ 1, 2, 3 ]
gap> list2[1] := 0;;  list2;
[ 0, 2, 3 ]
gap> list1;
[ 1, 2, 3 ]
```

That `Copy` returns the object itself if it is not a list or a record is consistent with this definition, since there is no way to change the original object *obj* by modifying *new*, because in fact there is no way to change the object *new*.

`Copy` basically executes the following code for lists, and similar code for records.

```
new := [];
for i  in [1..Length(obj)]  do
    if IsBound(obj[i])  then
        new[i] := Copy( obj[i] );
    fi;
od;
```

Note that `Copy` recursively copies all elements of the object *obj*. If you only want to copy the top level use `ShallowCopy` (see 5.12).

```
gap> list1 := [ [ 1, 2 ], [ 3, 4 ] ];;
gap> list2 := Copy( list1 );
[ [ 1, 2 ], [ 3, 4 ] ]
gap> list2[1][1] := 0;;  list2;
[ [ 0, 2 ], [ 3, 4 ] ]
gap> list1;
[ [ 1, 2 ], [ 3, 4 ] ]
```

The above code is not entirely correct. If the object *obj* contains a list or record twice this list or record is not copied twice, as would happen with the above definition, but only once. This means that the copy *new* and the object *obj* have exactly the same structure when view as a general graph.

```
gap> sub := [ 1, 2 ];;  list1 := [ sub, sub ];;
```

```
gap> list2 := Copy( list1 );
[ [ 1, 2 ], [ 1, 2 ] ]
gap> list2[1][1] := 0;;  list2;
[ [ 0, 2 ], [ 0, 2 ] ]
gap> list1;
[ [ 1, 2 ], [ 1, 2 ] ]
```

5.12 ShallowCopy

`ShallowCopy( obj )`

`ShallowCopy` returns a copy of the object *obj*. You may apply `ShallowCopy` to objects of any type, but for objects that are not lists or records `ShallowCopy` simply returns the object itself.

For lists and records the result is a **new** list or record that is **not identical** to any other list or record (see 3.8 and 5.3). This means that you may modify this copy *new* by assignments (see 3.5 and 5.2) or by adding elements to it (see 3.6 and 3.7), without modifying the original object *obj*.

```
gap> list1 := [ 1, 2, 3 ];;
gap> list2 := ShallowCopy( list1 );
[ 1, 2, 3 ]
gap> list2[1] := 0;;  list2;
[ 0, 2, 3 ]
gap> list1;
[ 1, 2, 3 ]
```

That `ShallowCopy` returns the object itself if it is not a list or a record is consistent with this definition, since there is no way to change the original object *obj* by modifying *new*, because in fact there is no way to change the object *new*.

`ShallowCopy` basically executes the following code for lists, and similar code for records.

```
new := [];
for i  in [1..Length(obj)]  do
    if IsBound(obj[i])  then
        new[i] := obj[i];
    fi;
od;
```

Note that `ShallowCopy` only copies the top level. The subojects of the new object *new* are identical to the corresponding subobjects of the object *obj*. If you want to copy recursively use `Copy` (see 5.11).

5.13 RecFields

`RecFields( rec )`

`RecFields` returns a list of strings corresponding to the names of the record components of

the record *rec*.

```
gap> r := rec( a := 1, b := 2 );;
gap> RecFields( r );
[ "a", "b" ]
```

Note that you cannot use the string result in the ordinary way to access or change a record component. You must use the *rec*.(*name*) construct (see 5.1 and 5.2).