Technische Hochschule Darmstadt
Fachbereich Informatik
Institut für Theoretische Informatik

SALIERI v2.0
Tutorial

Holger H. Hoos und Thomas Helbich
(Translated from German by Andrea Mayr and Holger H. Hoos)

Revision 1 — April 1995 (English version October 1997)

Important Notice

This tutorial is part of the documentation of the SALIERI *System, version 2.0 for MS-DOS. Most of its contents are perfectly applicable to the current version 2.0.7 of the* SALIERI *Language; however, there have been some minor modifications to the language. Furtheron, the new user interface of the* SALIERI *System, version 2.2 for MS-Windows is not covered here. We are currently working on the english documentation for the latest version of* SALIERI, *including an updated tutorial. Please check the* SALIERI *System and Language Homepage at* http://www.iti.informatik.th.darmstadt.de/AFS/CM/SAL *on the WWW for further information.*

Antonio Salieri (∗ Legnano 18th Aug.1750, † Vienna 7th May 1825) was a contemporary (and putative "'opponent"') of W.A. Mozart in Vienna, where he held among others the positions of "'Hofkapellmeister"' and (lateron) director of the academy of music. Salieri was an important tutor of his time, among his students we find such illustrous names as L. v. Beethoven, F. Schubert, F. Liszt and S. Sechter (who was on his part A. Bruckner's tutor). His main contribution as a composer was in the field of the opera of the Neapolitan school; however, he also composed numerous pieces of instrumental and sacral music. Despite their musical qualities, nowadays his compositions are being rarely performed and are largely unknown.

Contents

Conventions and Notations

In this document the following information is printed within a frame:

- Important information for the user, such as

  Note:

The correct function of this software is not guaranteed. The user is solely responsible for loss of data or other damage caused in connection with its application or operation.

- In example sessions and example files, user input is printed in using the "'typewriter"' font while the system responses are printed in italics such as

  Example: Example session

>`ver()` *This is Salieri - Version 2.0* >`1+2` *Result= 3* >

- Syntax of function calls and SALIERI operations, such as in

  Variation of the secondary parameters:

`cresc`(*seq*, int_1, int_2)	returns the sequence which is obtained from *seq*, by applying a linear intensity profile ranging from int_1 to int_2.
`accel`(*seq*, int_1, int_2)	returns the sequence which is obtained from *seq* by applying a linear tempo profile ranging from int_1 to int_2.

In addition the following special fonts are used:

- SALIERI, as the name of the system or the language, appears in small capitals.
- SALIERI operations and expressions are printed in "'typewriter"', e.g., `write()` or `[c1/4 d/8 g d/4 e]`.
- The parameters of SALIERI operations are set in italics, e.g., `length(`*string*`)`.
- Names of operating system objects, such as file names, paths and environment variables, are printed in "'typewriter"', e.g., `SALIERI.EXE` or `LIBS\*.SAL`.

1 Introduction

The SALIERI System is an interactive system for structure-oriented generation, manipulation, and analysis of music. This manual is part of the documentation of SALIERI version 2.0 for MS-DOS.

Essentially, SALIERI consists of two components: the interpreter of the SALIERI Language and a command-line user interface. Musical objects can be saved as MIDI files; this facilitates communication with software for playback or musical notation. When SALIERI is used within a multitasking environment, musical objects can be played directly from the SALIERI System by means of a special background process (demon).

The SALIERI Language is a universal and essentially functional musical programming language which specifically supports the structure-oriented representation and manipulation of single and multi-voice musical material. In order to represent musical objects, a simple, perception related tone model is used, which comprises essential parts of the conventional musical notation in a symbolic notation. Besides the musical object types SALIERI supports the common basic types for truth values, numerical values and character strings. In order to realise more complex data structures, lists over objects of arbitrary and possibly different types can be formed.

SALIERI supports about 100 predefined functions; these functions realise operations on the different data types as well as control structures and system commands. Musical objects can be manipulated using a large number of variator and combinator functions. The SALIERI Language further supports user-defined functions, it thus facilitates the adaption of the system according to specific requirements of the user. Virtually every musical function that can be algorithmically stated can be realised as a user-defined function based on the predefined variator and combinator operations.

The main motivation behind the SALIERI System is to realise the basic principles and aspects of structure-oriented presentation and manipulation of music in the form of an interactive system. The experimental realisation at hand is certainly not able to achieve all that may be desired of the system; the user interface is quite primitive, and there is definitely some need of improvement with respect to error handling. A further development of the system (in particular with respect to the user interface) is planned and will be significantly influenced by the experiences gathered from using the system.

Note:

As the SALIERI System is considered to be an experimental realisation, system failures or loss of data may occur. Therefore it is highly recommended

- to save the user data within SALIERI (workspace) frequently (by means of the `sync()` command, see section 3.4);
- to save the user data of all active programs before SALIERI is called when multitasking systems (such as MS-Windows) are used
- to backup all important data from the system SALIERI is running on (e.g. on diskettes).

The correct function of this software is not guaranteed. The user is solely responsible for loss of data or other damage occuring in connection with its application or operation.

2 Basics

2.1 Command-line Input

The user interface of SALIERI is command-line oriented (like the user interface of MS-DOS). Inputs are terminated by *Enter* and must not exceed the end of the line. Using the ↑, ↓ keys, previous input lines can be recalled, The ←, →, *Home, End, Del* and *Backspace* keys (with the usual functionality) serve to edit the command line. By pressing *Esc* the current command line is (irrevocably!) cleared.

All inputs (confirmed with *Enter*) of a session are recorded in the file `SALIERI.LOG`, this file is created anew at the beginning of each session, with the previous contents being lost.

Each input confirmed with *Enter* is processed by the SALIERI Interpreter. There is a basic distinction between commands and expressions. Commands are executed, and may cause system responses such as confirmations or error messages. Expressions, however, are evaluated, with the result being put out in the form *Result=....* If an error occurs during the evaluation, an error message (and not a result) is given.

The function `it()` returns the last result of an expression evaluated by SALIERI. Immediately after the start of the system `it()` is not defined. :-)

Example:

```
>ver()
This is Salieri - Version 2.0
>1+2
Result= 3
>1+x
Error in command-line: Unknown identifier: x
>it()
Result= 3
>
```

Attention:

Functions without parameters and commands (such as `exit` or `ver`) have to be called using the syntax *name*`()` (see section 6.2).

2.2 SALIERI Files

Besides the interactive command-line operation, the SALIERI System can execute SALIERI program files. The command `exec("`*filename*`")`, which executes the program file *filename*`.SAL` (from the current working directory) serves this purpose.

SALIERI program files contain a series of commands, which are separated from eachother by "';"' respectively. Blank lines and additional space are ignored (as usual), comments are written in the form `(* ... *)` and they are completely ignored when programs are executed; they can occur arbitrarily nested. The commands in a SALIERI program file are executed just as if they had been invoked from the command-line in the respective order.

Example: Program file `DEMO1.SAL`

```
x := 10+2;              (* Object x is created *)
stat();                 (* Display the system state *)
write();                (* write blank line *)
write("Result: ",       (* Line break permitted!  *)
       2*(x-1));
```

Example: Executing `DEMO1.SAL`

>`exec("demo1")`
Executing demo1.sal ...
1 object in workspace:

x

229008 bytes of memory free

Result: 22
>

Note:

1. In program files, commands may comprise an arbitary number of lines (including blank lines).
2. The last command of a program file can also be an expression; however, this usage should be avoided.
3. The last command of a program file need not be succeeded by a semicolon; it is, however, highly recommened to use it (in order to prevent errors in case the file is extended later).

Program files can be used to implement function libraries. In this case the program generally consists of a number of assignments (see section 3.1) and, possibly, additional commands for initialization.

A correct program file can be easily created from the log file `SALIERI.LOG` by deleting all lines that consist of expressions. Before it is executed, however, it has to be renamed to *filename*`.SAL`.

3 Workspace and Objects

Every object has a name, a type and a value. All objects are stored in the workspace and arranged in lexicographical order with respect to their names (except for the internal functions, see section 6.5). Object names always start with a letter or an underscore ("'_'"), followed by an arbitrary number of letters, digits, or underscores. SALIERI is case sensitive, i.e., capital and small letters are considered to be different.

Example: The names `x`, `x1`, and `x_new` are valid object names, whereas `1x` is not permitted as an object name.

Please note:

Objects, the names of which start with an underscore (such as `_x`), are so-called system objects. Such objects are generally not shown by output commands such as `statAll()` (see section 3.2) and should only be applied by experienced users. Unexperienced users should neither create nor alter or delete system objects.

3.1 Creating and Deleting Objects

An object is created by assigning a value to a name which is not yet in use. The type of the object implicitly results from the type of the assigned value.

Syntax of assignments

name := *expression*

name is a valid name and *expression* is an arbitrary expression. The most simple expressions are constants and names of objects. Moreover, expressions can be built up from other expressions by applying functions or prefix operators or by linking them with infix operators. Finally expressions can be put into parentheses.

Syntax of expressions

If c is a constant, `x` an arbitrary object name, `f` the name of a n-place function, "'□"' a prefix operator, "'◇"' an infix operator and if $exp_1, \ldots, exp_n$ are valid expressions, the following are valid expressions as well:

- c
- `x`
- `f(`$exp_1, \ldots, exp_n$`)`
- $\Box\ exp_1$
- $exp_1 \diamond exp_2$
- (exp_1)

The constants, functions and operators available in Salieri are introduced in the following sections.

If a value is assigned to an object which exists already, the old value is automatically replaced by the new one while the type is automatically adjusted.

Example: Assigning values and creating Objects

```
>x_1 := 1
>x_1
result= 1
>x_1 := "hello"
>x_1
Result= "hello"
>
```

The commands `delete(`*name*`)` and `deleteAll(`*templStr*`)` are used to delete objects, the first of which deletes the object called *name* from the workspace (in case it exists, otherwise an error occurs!). The command `deleteAll(`*templStr*`)` deletes all objects, the names of which match the template *templStr*. This template is a character string which may contain the wildcard character "'`*`"' at arbitrary positions. This character matches arbitrary character strings (including the empty string, i.e. no character at all!) at the corresponding position of the object name.

Example: The template `"any*_*1"` matches the object names `any_1`, `anyNewX_a1` and `anyNewX_main_a1`, but not `any`, `Any_1` or `anyNewX_a2`.

Example: Deleting objects

```
>stat()
3 objects in workspace:

x_1 x_2 y

229008 bytes of memory free
>deleteAll("x*")
Deleting objects x*:

x_1
x_2
>stat()
1 object in workspace:

y

229008 bytes of memory free
>
```

Comment: When objects are deleted, the corresponding memory is generally not immediately released (this effect is due to the internal object and memory management).

3.2 Inspecting objects

The command `stat()` lists all the names of the objects located in the workspace. It also informs about the memory still available. Moreover, information about all objects, the names of which fit the template *templStr*, can be given by means of the command `statAll(`*templStr*`)`.

The values of certain objects can be inspected using the commands `write(`*name*`)` and `writeAll(`*templStr*`)`.

The function `type(`*expr*`)` yields the type of the expression *expr* as a character string. In particular *expr* can be an object name.

The function `exists(`*name*`)` is used to check whether an object exists in the workspace at all. It returns `TRUE`, if an object with the name *name* is defined, otherwise `FALSE`.

Example: Inspecting of objects

```
>x_1:="hello"
>exists(x_1)
Result= TRUE
>type(x_1)
Result= "string"
>write(x_1)
hello
>
```

3.3 Saving and loading the workspace

A copy of the current workspace can be saved in the file *filename*`.WSP` (located in the current directory, if no path is specified) by means of the command `save(`*filename*`)`. In order to reload the contents of such a workspace file into the current workspace, the command `load(`*filename*`)` is called. In doing so the contents of the workspace saved in *filename*`.WSP` is added to the current workspace. If an object name was used in the current workspace as well as in the workspace c opy saved, the respective object is overwritten in the current workspace, when the workspace file is loaded.

3.4 Persistence

If not specified otherwise when starting a SALIERI session (see section ??), the workspace is persistent. This means that the workspace is saved (in the file `SALIERI.WSP`) at the end of every session; this workspace copy is reloaded when the SALIERI System is restarted, so that the previous contents of the workspace are preserved between two successive SALIERI sessions.

This mechanism, however, only works if the SALIERI session was terminated properly using the `exit()` command. In order to guarantee a better protection against the loss of the contents of the workspace caused by a failure of the system, the command `sync()` can be used to update the workspace file `SALIERI.WSP` anytime during a SALIERI session. This command has the same effect as the command `save("SALIERI.WSP")`.

4 Simple Object Types

Although the SALIERI System was mainly designed for manipulating musical structures, it also covers the common simple data types. These data types are introduced and explained in the following sections.

4.1 Truth values

Truth values are implemented by the object type `"bool"`. Besides the constants `TRUE` and `FALSE` the common logic operations `not, and, or` are available within boolean expressions (i.e. expressions with the result type `"bool"`):

Boolean operations

`not` *bool*	logic negation
bool `and` *bool*	logic conjunction
bool `or` *bool*	logic disjunction

In boolean expressions, `not` has a higher precedence than `and`, which in turn has precedence over `or`. *Example:* If `x,y` are objects of type `"bool"`, "'`x and not y`"', "'`x and y or not x and not y`"' are valid boolean expressions, the second of which is equivalent to "'`(x and y) or ((not x) and (not y))`"'.

Example: Boolean Operations

```
>x := TRUE
>y := FALSE
>x and y
Result= FALSE
>x or y
Result= TRUE
>
```

4.2 Integer

The type `"int"` implements integer numbers between -32768 and 32767. The constants are written as common numerical values with a sign ("'+"' can be omitted, i.e. $+256 = 256$). The common arithmetic operations are defined (addition, subtraction, multiplication, integer division, division remainder) on objects of this type, in addition functions are defined for computing absolute values and generating uniform random numbers in a given interval.

Integer operations

`+`*int*	plus sign
`-`*int*	minus sign
int `+` *int*	addition
int `-` *int*	subtraction
int `*` *int*	multiplication
int `div` *int*	division by integers (`7 div 3 = 2`)
int `mod` *int*	division remainder (`7 mod 3 = 1`)
`abs(`*int* `)`	absolute value
`rndUniform(`int_1`,`int_2`)`	returns random number i with $int_1 \leq i \leq int_2$

The following precedence rules apply to integer expressions: Signs have precedence over `*,div,mod`, which have higher precedence than `+,-`. Apart from that integer expressions are evaluated from left to right , i.e. "'`-1 + 11 div 5 mod 3`"' corresponds to"'`(-1) + ((11 div 5) mod 3)`"' and results in `1` when evaluated.

Attention:

Exceeding the numerical range for integer operations (e.g., when addition or multiplication is used) does not cause any error messages to occur, but generaly leads to incorrect results.

Example: Integer operations

```
>n := 10
>(2*n+3) div 10
Result= 2
>rndUniform(-2*n,+2*n)
Result= 0
>rndUniform(-2*n,+2*n)
Result= -16
>
```

4.3 Reals

In SALIERI floating-point numbers are realized by the type `"real"`. The constants are written using the decimal point and signs ("'+"' can be omitted, i.e. $+12.3 = 12.3$). For real values, the common arithmetic operations (addition, subtraction, multiplication, division), functions for computing the absolute value and for generating equally distributed random numbers, and standard mathematical operations (see examples below) are available. Furtheron, floating-point numbers can be converted into integers by rounding up or down.

Explicitly converting integers into floating-point numbers can be achieved by multiplying them with `1.0`. Furtheron, integers can be used wherever floating-point numbers are expected; this causes an automatic type conversion (as in the expression `5/2`, which evaluates to `2.5`). Floating-point numbers are always displayed by the SALIERI System using 8 digits after the decimal point (with the last digit being rounded). In general, however, real number calculations are performed internally using a much higher precision.

Real operations

`+`*real*	plus sign
`-`*real*	minus sign
real `+` *real*	addition
real `-` *real*	subtraction
real `*` *real*	multiplication
real `/` *real*	division
`abs(`*real*`)`	absolute value
`rnd()`	returns random number r with $0 < r < 1$
`floor(`*real* `)` $\rightsquigarrow$ *int*	rounding down (truncate)
`ceil(`*real* `)` $\rightsquigarrow$ *int*	rounding up
`root(`$real_1$`,` $real_2$ `)` $\rightsquigarrow$ *real*	root of $real_2$ to the base of $real_1$
`root(`*real*`)` $\rightsquigarrow$ *real*	square root
`lexp(`$real_1$`,` $real_2$ `)` $\rightsquigarrow$ *real*	$real_1$ raised to the power of $real_2$
`exp(`*real*`)` $\rightsquigarrow$ *real*	the Euler number raised to the power of *real*
`log(`$real_1$`,` $real_2$ `)` $\rightsquigarrow$ *real*	logarithm of $real_2$ to the base of $real_1$
`log(`*real*`)` $\rightsquigarrow$ *real*	natural logarithm
`sin(`*real*`)` $\rightsquigarrow$ *real*	sine (in circular measure)
`cos(`*real*`)` $\rightsquigarrow$ *real*	cosine (in circular measure)
`tan(`*real*`)` $\rightsquigarrow$ *real*	tangent (in circular measure)
`arctan(`*real*`)` $\rightsquigarrow$ *real*	arcustangent (in circular measure)

Precedence rules: Signs have a precedence over `*,/`, which have a precedence over `+,-`. Apart from that the expressions are evaluated from left to right.

Example: Real operations

```
>5/2*3
Result= 7.50000000
>floor(it())
Result= 7
>ceil(-1.2)
Result= -1
>1.0*it()
Result= -1.00000000
>5/7
Result= 0.71428571
>it()*100
Result= 71.42857143
>
```

Attention:

- Floating-point arithmetics are not always exact. Especially when large numbers are involved, inaccuracies in the last digits can occur.
- Generally, exceeding the numerical range for floating-point operations can lead to severe error conditions which might result in crashes of the SALIERI System (with the entire contents of the workspace being lost!).

4.4 Character Strings

In SALIERI the type `"string"` implements character strings. The constants of this type are written as `"..."`, where ... represents an arbitrary character string, which does not contain any double quotes. In addition to the basic operation of string concatenation functions for deleting, inserting, and extracting substrings are available. Besides this there are functions for determining the length of a character string (in characters) as well as for locating substrings. The positions within a character string with a length of n are numbered from 1 to n so that in `"J.S.Bach"` (length 8) the character `"S"` is located at position 3. Basically, character strings can reach a length of up to 32767 characters.

String operations

string `+` *string*	concatenation
`infix(`*string*, int_1, int_2`)`	returns a section of *string*, which starts at int_1 and has a length of int_2; for positive int_2 it is counted to the right for negative values it is counted to the left. int_2 can be omitted — then a section of the length 1 is returned as standard.
`ins(`$string_1$, $string_2$, *int*`)`	returns a character string, which is obtained by inserting $string_2$ into $string_1$ directly in front of position *int* *int* has to be between 1 and `length(`$string_1$`)+1`.
`del(`*string*, int_1, int_2`)`	returns a character string, which is obtained from *string* by deleting int_2 characters, beginning with the character at position int_1, for positive int_2 it is counted to the right, for negative values it is counted to the left. int_2 can be ommitted — then only one character is deleted as standard.
`length(`*string*`)` $\rightsquigarrow$ *int*	returns the length of *string* in characters.
`posNext(`$string_1$, $string_2$, *int*`)` $\rightsquigarrow$ *int*	returns the position where $string_2$ occurs next in $string_1$, with the search starting from position *int* to the right; if there is no such occurrence, the value 0 is returned. *int* can be ommitted — then the search starts from the first position as standard.

Notation: `#`*string* can be written equivalently for `length(`*string*`)`. The operator "'`#`"' has precedence over "'`*`"' or "'`/`"'. `infix(`*string*, int_1, int_2`)` can be abbreviated by *string* (int_1,int_2) where int_2 is optional. However, this form of `infix` can only be used for existing string objects, not for constants.

Note:

- The functions `infix` and `del` also accept length values int_2 exeeding the last (or the first) position of the character string. In this case, the length is interpreted as if it just reached the beginning (or end) of the string.
- If an invalid position `int` or $\texttt{int}_1$ is given, when `ins` or `del` are called, an error condition occurs.
- If `infix` or `posNext` are called with invalid positions int_1 or int, `""` or, `0` is returned as result.

These operations are also defined for other object types which have serial structure (note sequences and lists) — they are therefore called serial operations. Due to the manifold applications of these serial operations, they are illustrated in detail in the following example.

Example: Serial operations on character strings

```
>name := "A.Salieri";
>piece := "piano concerto B major";
>name + ", " + piece
Result= "A.Salieri, piano concerto B major"
>#it()
Result= 31
>infix(name, 3,3)
Result="Sal"
>name(3,3)
Result= "Sal"
>name(5)
Result="l"
>infix(name, 3,50)
Result= "Salieri"
>infix(name, 3,-3)
Result= "A.S"
>infix(name, 3,-10)
Result="A.S"
>infix(name, 3,0)
Result= ""
>posNext(piece,"er")
Result= 6
>posNext(piece, "er", 7)
Result=12
>posNext(piece, "er", 13)
Result= 0
>del(piece, 15, 8)
Result= "piano concerto "
>del(name, 2, -2)
Result="Salieri"
>ins(piece, "in ", 16)
Result= "piano concerto in B major"
>ins(name, " (1750-1825)", #name+1)
Result= "A.Salieri (1750-1825)"
>ins(name, "(1750-1825)", 12)
Error in command-line: Wrong startposition
(occurred while executing function "ins")
>
```

4.5 Relational Operators

In SALIERI six relational operators are available: ”‘`=`”’ and ”‘`!=`”’ (equality and inequality) are defined for all simple and musical object types (see Section 5); moreover, the operators ”‘`<`”’, ”‘`>`”’, ”‘`<=`”’ and ”‘`>=`”’ exist for numerical values and character strings (for character strings, these are based on the lexicographical ordering). All relational operators are infix operators with the result type`"bool"` and have precedence over the logical operators, but a lower precedence than all other operators (for this reason the expressions ”‘`(((2*x)+5) > 10) and (not (x > 5))`”’ and ”‘`2*x+5 > 10 and not` `x > 5`"' are equivalent).

Note: Comparisons can only be carried out between objects of the same type, in particular comparisons such as "'`1/4 = "hello"`"' are invalid (and do not return `FALSE`!). Comparisons between integers and floating-point numbers are permitted due to the automatic conversion from *int* into *real*.

Example: Relational operations

```
>5 < 7
Result=TRUE
>"abc" = "abcd"
Result= FALSE
>"abc" < "abcd"
Result= TRUE
>"abc" >= "abd"
Result= FALSE
>FALSE != 1
Error in command-line: Type mismatch
(occurred while executing function "!=")
>
```

5 Musical Object Types

In the first place the SALIERI System was designed to facilitate the generation and manipulation of musical material. Musical structures are represented on the score level, i.e., as notes and rests. SALIERI provides two basic musical object types: Note sequences (`"sequence"`) for representing one-voice music and note segments (`"segment"`) for representing multi-voice material.

In the following sections, first the constants for note sequences and note segments are introduced, followed by the basic operations on musical objects, and finally other musical operations are discussed.

5.1 Sequence Constants

In the SALIERI System, a musical tone is described by 5 parameters: these are pitch and relative duration (the so-called main parameters), intensity, instrument, and tempo (the secondary parameters).

A complete tone description is given by "`M=`m `T=`t `I=`i $qr{*}e/d$ ", where m, t, i, q, r, e, d have the following interpretation:

m tempo in quarter notes per minute (metronome number), integer between 1 and 255

t instrument as a MIDI instrument code, integer between 0 und 127

i (relative) intensity as a MIDI parameter, integer between 0 und 127 (logarithmic scale!)

q pitch-class, either one of the note names `c,d,e,f,g,a,h`, if necessary, directly followed by accidentals `#` or `&`; or "`_`" for rests, in addition it is also possible to use the chromatic names `cis,dis,fis,gis,ais` or the Italian note names `do,re,mi,fa,sol,la,si` (see also system settings, `set("noteNames",...)`)

r register (octave), integer between -4 and +6

e numerator of the relative duration, integer between 1 und 128

d denominator of the relative duration, integer between 1 and 128

In addition the duration, represented by e/d, can be modified by adding dots. Dotted and double-dotted time values of notes can thus be represented. The dots have to be placed directly after the duration parameters.

Example: "`M=100 T=0 I=80 a1*1/4`", "`M=220 T=18 I=4 _*5/27`", "`M=118 T=0 I=127 fis&*3/8`", and "`M=80 T=10 I=40 c*1/8.`" are complete tone descriptions.

Note sequence constants are sequences of tone descriptions denoted as `[`n_1 n_2 ...`]`. For the tone descriptions n_1, n_2 etc., the pitch-class must always be specified, all other parameters can be omitted. In that case, the missing parameters are assumed to be identical to the corresponding parameters of the previous note. (Exception: the relative duration — including dots — will only be copied, if neither e nor d are specified. For incomplete durations such as `/8` or `*2` are given, the missing parameters is assumed to be equal to one.) If the first note of a sequence is incompletely specified, the following default values for missing parameters will be used:

- for the octave $r = 1$
- for the duration $e/d = 1/4$
- for the tempo $m = 100$
- for the instrument $t = 0$ (which normally corresponds to the piano)
- for the intensity $i = 80$ (which normally corresponds to mezzoforte)

The default values for tempo, instrument, and intensity are part of the system settings and can be modified.

The output format for note sequences depends on a number of system settings which can be modified by the user.

Example: Output format of note sequences

```
>set("noteNames","diatonic")
>[I=80 a/8 h c2/4 I=96 do h1 cis2/2]
Result= [ I=80 M=100 T=0 a1/8 h1/8 c2/4 I=96 c2/4 h1/4 c#2/2 ]
>set("seqDispFormat","short")
>set("noteNames","chromatic")
>[do& h1 cis2/2]
Result= [ I=80 M=100 T=0 c&1/4 h1 cis2/2 ]
>set("dispTags","off")
>set("noteNames","DoReMi")
>[do& h1 cis2/2]
Result= [ do&1/4 si1/4 do#2/2 ]
>
```

The relational operators "`=`" and "`!=`" can be applied to note sequences, with two note sequences being equal if they have the same length and if all pairs of corresponding notes have identical main parameters (pitch and relative duration).

Note sequences can basically reach a length from up to 32767 notes. Currently, however, note sequences with more than 1000 notes are quite impractical (due to long processing times and memory requirements).

Note:

- Generally, rests are not automatically merged.
- `c#`, `d&` and `cis` are considered to be different pitch-classes. Consequently, for example, evaluating `[c#]=[cis]` yields *FALSE*.

5.2 Segment Constants

A segment consists of a number of note sequences, the voices. All voices of a segment start at the same time and proceed in parallel. The constants of the type `"segment"` can be written in two different ways.

In simple polyphonic form they are denoted as $\{seq_1,\ seq_2,\ \ldots\}$, with seq_1 etc. being note sequences (as specified above).

Example: {`[c d e], [e f g]`} and {`[c d e]`} represent segments in simple polyphonic form.

In addition it is possible to write segments as chord sequences in the form `[`$a_1\ a_2\ \ldots$`]`, with a_1 etc. being either tone descriptions (see above) or chord descriptions of the form $\{n_1,\ n_2,\ \ldots\}$ (where n_1 etc. are tone descriptions).

Note: Within a chord, tone parameters which are not specified are copied from the last note, so that `[{c2/8, e, g}]` is interpreted as `[{c2*1/8, e2*1/8, g2*1/8}]`.

Example: `[{c,e,g} {h0,d1,g}]` and `[{c,e,g}]` are segments in chord sequence form.

Note segments are always automatically transformed into simple polyphonic normal form by the

Salieri System. In this form a segment consists of a series of parallel note sequences with identical duration. In order to obtain this form, chords are distributed into seperate voices (where the first note of a chord description will appear in the first voice). Then all voices are filled with rests as required.

Example: Transformation in simple polyphonic normal form

```
>{[g-1/4], [_/4 h0/2]}
Result= SEGM([ I=80 M=100 T=0 g-1/4 _/2 ],[ I=80 M=100 T=0 _/4 h0/2 ])
>
```

Note:

- The combination of simple polyphonic and chord sequence form in segment constants (such as `{[{c,e,g} {h0,d1,g}]}` or `[{[c e g]}]`) is not permitted.
- The order of the voices in a segment is significant:
  $\{seq_1,\ seq_2\} \neq \{seq_2,\ seq_1\}$
  (The same holds for the order of notes in a chord, as this determines the assignment to the voices).

The relational operators "`=`" and "`!=`" can be applied to note segments, with two segments being equal, if they consist of equal voices in the same order.

The (hypothetical) limit for the number of voices within a segment is 32767. However, only 16 completely independent voices can be used for MIDI output (if more voices are used, MIDI channels are multiply assigned which can result in problems with instrument settings).

5.3 Basic Operations on Sequences

The serial functions `+`, `infix`, `ins`, `del`, `length` and `posNext` are defined as basic operations (compare Section 4.4 on strings; note: when locating subsequences, `posNext` compares only the main parameters!). They are defined as for character strings, with the "basic objects" being notes instead of characters. In addition a function `retro(`*seq*`)` is available, which yields the temporal inversion of a given note sequence. This function can also be applied to segments, where it is applied to each voice individually.

Example: Serial operations on note sequences

```
>sq := [e2/16 d c h1 c2/8 a1 f2/4]
>#sq
Result= 7
>infix(sq, 3,4)
Result= [ I=80 M=100 T=0 c2/16 h1/16 c2/8 a1/8 ]
>posNext(sq, [c2/16], 1)
Result= 3
>retro(sq)
Result= [ I=80 M=100 T=0 f2/4 a1/8 c2/8 h1/16 c2/16 d2/16 e2/16 ]
>
```

Transposition and (temporal) expansion are realized by the basic variator functions `transp` and `expand`:

Main parameter variation:

`transp(`*seq*`,` *int*`)`	returns a sequence obtained from *seq* by transposing it upwards by *int* semitones.
`expand(`*seq*`,` int_1`,` int_2`)`	returns a variant of *seq* which is temporally "expanded" (or "compressed") by the factor int_1/int_2; to be more exact, each relative duration of the form e/d is replaced by $(e * int_1)/(d * int_2)$.

Downward transpositions can be achieved by specifying a negative number *int* of semitones. The functions `transp` or `expand` only alter the pitches or the relative durations of the given sequence respectively, while leaving the other parameters unaffected.

Example: Transposition and expansion

```
>sq := [e2/16 d c h1 c2/8 a1 f2/4]
>transp(sq, +7)
Result= [ I=80 M=100 T=0 h2/16 a2/16 g2/16 f#2/16 g2/8 e2/8 c3/4 ]
>transp(sq, -5)
Result= [ I=80 M=100 T=0 h1/16 a1/16 g1/16 f#1/16 g1/8 e1/8 c2/4 ]
>expand(sq, 1,2)
Result= [ I=80 M=100 T=0 e2/32 d2/32 c2/32 h1/32 c2/16 a1/16 f2/8 ]
>expand(sq, 2,1)
Result= [ I=80 M=100 T=0 e2/8 d2/8 c2/8 h1/8 c2/4 a1/4 f2/2 ]
>
```

The basic operations for varying the secondary parameters (intensity, tempo and instrument) are `cresc`, `accel` and `instr`. These are defined as follows:

Variation of the secondary parameters:

`cresc`(seq, int_1, int_2)	returns the sequence which is obtained from *seq* by combining it with a linear intensity profile from int_1 to int_2.
`accel`(seq, int_1, int_2)	returns the sequence which is obtained from *seq* by combining it with a linear tempo profile from int_1 to int_2.
`instr`(seq, int)	returns the sequence which is obtained from *seq* by setting the instrument to *int*.

Using these functions one should keep in mind that certain restriction apply to the values for intensity, tempo and instrument (see above). Furtheron, by applying the functions `cresc` (or `accel`) any previously existing intensity (or tempo) profile is overwritten.

Example: Operations for varying secondary parameters

```
>sq := [e2/16 d c h1 c2/8 a1 f2/4]
>sq := cresc(sq, 32,96);          (* p -> f *)
>sq := accel(sq, 60,120);         (* MM=60 -> MM=120 *)
>sq := instr(sq, 6);              (* cembalo *)
>sq
Result= [ I=32 M=60 T=6 e2/16 I=41 M=68 d2/16 I=50 M=77 c2/16 I=59 M=85 h1/16 I=68
M=94 c2/8 I=77 M=102 a1/8 I=86 M=111 f2/4 ]
>
```

The functions `transp`, `expand`, `cresc`, `accel`, and `instr` can also be applied to segments, in this case the corresponding operation is applied to each single voice.

5.4 Basic operations on segments

As already mentioned a note segment consists of a series of sequences, the voices. The function `parts`(*segm*) returns the number of voices, segment *segm* consists of. `#`*segm* can be written shorthand for `parts`(*segm*). The voices of a segment are numbered $1, \ldots, n$; the function `part`(*segm*, *int*) extracts voice *int* of *segm* (or the empty sequence `[]`,if $int > n$ or $int < 1$).

Example: Basic operations on note segments (1)

```
>sg := {[e2/16 d c h1 c2/8 a1 f2/4], [a0/8 a1*3/16 h/16 c2 h1 a g f e]}
>#sg
Result= 2
>part(sg,2)
Result= [ I=80 M=100 T=0 a0/8 a1*3/16 h1/16 c2/16 h1/16 a1/16 g1/16 f1/16 e1/16 ]
>
```

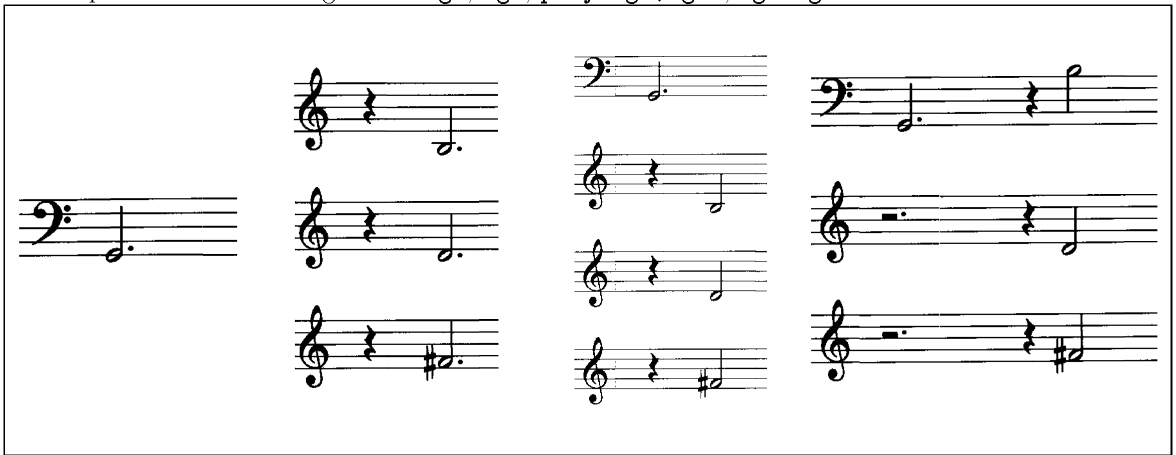
The two most important operations on note segments are `+` and `poly`. The function `poly`(s_1, ... ,s_n) combines s_1, ... ,s_n to a new segment, in which all the single voices of s_1, ... ,s_n (starting at the same time) proceed in concurrently. By filling up the individual voices with rests at the end, it is guaranteed that all voices of the new segment have the same duration. Here, s_1, ..., s_n represent arbitrary segments or sequences (in this special case sequences are treated like the corresponding one-voice segments!). $\{\ s_1, \ldots, s_n\}$ can also be used as an abbreviation for `poly(`$s_1, \ldots, s_n$`)`.

By means of the operator `+` segments can be (temporally) concatenated. For s_1+s_2, this is realised by concatenating the first voice of s_2 with the first voice of s_1 which results in the first voice of the new segment etc. (Note: All voices of a segment have basically the same duration!). If the number of voices differs from s_1 and s_2, first of all it will be compensated for by filling up one of them with empty voices.

Example: Basic operatios on note segments (2)

```
>sg1 := {[g-1*3/4]}
>sg2 := [_/4 {h0/2,d1,f#}]
>poly(sg1,sg2)
Result= SEGM([ I=80 M=100 T=0 g-1*3/4 ],[ I=80 M=100 T=0 _/4 h0/2 ], [ I=80 M=100 T=0 _/4 d1/2 ],[ I=80 M=100 T=0 _/4 f#1/2 ])
>{[c],{[e],[g]}}
Result= SEGM([ I=80 M=100 T=0 c1/4 ],[ I=80 M=100 T=0 e1/4 ],[ I=80 M=100 T=0 g1/4 ])
>sg1+sg2
Result= SEGM([ I=80 M=100 T=0 g-1*3/4 _/4 h0/2 ], [ I=80 M=100 T=0 _*3/4 _/4 d1/2 ], [ I=80 M=100 T=0 _*3/4 _/4 f#1/2 ])
>
```

Basic operations on note segments: `sg1`, `sg2`, `poly(sg1,sg2)`, `sg1+sg2`



5.5 Musical descriptor functions

Descriptor functions supply information about musical objects. The following functions of this kind are available in SALIERI:

Descriptor functions:

`isRest`(*seq*, *int*) $\leadsto$ *bool*	returns `TRUE`, exactly if the note at position *int* of *seq* is a rest, otherwise `FALSE` . The parameter *int* can be omitted, then position 1 is assumed as a default.
`getPitch`(*seq*, *int*) $\leadsto$ `LIST`(int_1, int_2, int_3)	returns the pitch at position *int* of *seq* as a list of pitch-class (int_1 (compare section 8.1); where $0, 1, \ldots$ represent $c, c\#, \ldots$) and octave (int_2); int_3 represents pitch alterations in semitones, where negative values represent flats, positive values sharps. If *seq* begins with a rest, -1 will be returned as the pitch-class value. The position parameter *int* can be omitted, then position 1 is assumed as a default.
`getPitchTonal`(seq_1, seq_2, *int*) $\leadsto$ `LIST`(int_1, int_2)	returns the relative pitch of a note at position *int* of the the sequence seq_1 with respect to an arbitrary scale seq_2. The parameter *int* can be omitted, then position 1 is assumed as a default. int_1 represents the scale position of the note with respect to scale scale seq_2, where the first note of the scale is represented by 0, the second by 1. Lower notes, than those defined in the sequence scale, are represented by the respective negative values. int_2 specifies chromatic alterations in semitones.
`ambitus`(*seq*) $\leadsto$ *int*	returns the ambitus (i.e. the interval between the highest and the lowest note) of the sequence *seq* in semitones.
`duration`(*seq*) $\leadsto$ `LIST`(int_1, int_2)	returns the relative duration int_1/int_2 of the sequence *seq* (as a list numerator,denominator); the relative duration of a sequence is the sum of the relative durations of the individual notes and rests.
`time`(*segment*) $\leadsto$ *real*	returns the absolute duration of the segment *segment* in seconds; in contrary to the relative duration, the tempo profile is taken into account here. This functions can also be applied to sequences.
`getIntens`(*seq*, *int*) $\leadsto$ *int*	returns the intensity of the note at position *int* of the sequence *seq*.
`getTempo`(*seq*, *int*) $\leadsto$ *int*	returns the tempo value for the note at position *int* of the sequence *seq*.
`getInstr`(*seq*, *int*) $\leadsto$ *int*	returns the instrument value for the note at position *int* of the sequence *seq*.

By means of the function `getPitch`, intervals can easily be determined. The same holds for duration ratios ("rhythmic intervals") when the function `duration` is used.

Example: Musical Descriptor functions

```
>sq := [M=100 _/8 g h g d2 g1 g&2]
>isRest(sq,1)
Result= TRUE
>getPitch(sq,#sq)
Result= LIST(6,2,-1)
>getPitchTonal(sq,[c1 d e f g a h],#sq)
Result= LIST(11,-1)
>ambitus(sq)
Result= 11
>duration(sq)
Result= LIST(7,8)
>time(sq)
Result= 2.09999990
>
```

5.6 Musical variator functions

SALIERI provides two functions for varying the melodic profile of segments or sequences: `invert` and `intvExpand`; these implement an inversion or expansion (or a compression) of the interval structure. These functions are defined in the following way:

Variation of interval sequences:

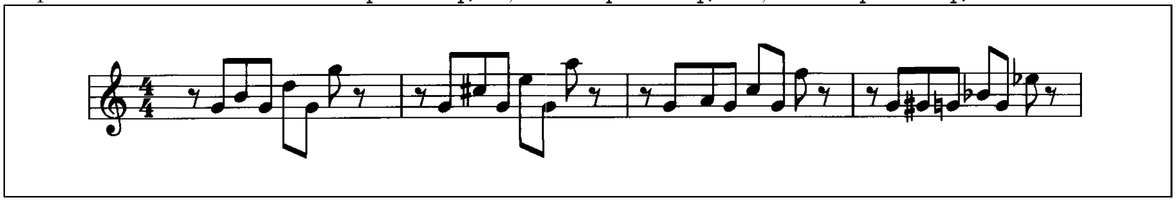
`invert(`*seq*`)`	returns the sequence which is obtained by inverting all intervals in *seq*, where the first note and all interval sizes are preserved, while the interval directions are all reversed.
`intvExpand(`*seq*, *int*`)`	returns the sequence which is abtained from *seq*, by increasing all interval sizes by *int* semitones; the intervals directions are preserved and for negative *int* values, minor seconds will not be further reduced.

Rests are basically ignored applying these interval operations; for negative parameter values of `intvExpand` the interval sizes will be reduced by the respective number of semitones.

Example: Variation of the interval sequences

```
>sq := [_/8 g h g d2 g1 g2]
>invert(sq)
Result= [ I=80 M=100 T=0 _/8 g1/8 e&1/8 g1/8 c1/8 g1/8 g0/8 ]
>intvExpand(sq,2)
Result= [ I=80 M=100 T=0 _/8 g1/8 c#2/8 g1/8 e2/8 g1/8 a2/8 ]
>intvExpand(sq,-2)
Result= [ I=80 M=100 T=0 _/8 g1/8 a1/8 g1/8 c2/8 g1/8 f2/8 ]
>intvExpand(sq,-4)
Result= [ I=80 M=100 T=0 _/8 g1/8 g#1/8 g1/8 h&1/8 g1/8 e&2/8 ]
>
```

Expansion of intervals: `intvExpand(sq,2)`, `intvExpand(sq,-2)`, `intvExpand(sq,-4)`



Besides the function **`transp`** two additional transposition functions are available: **`transpTonal`** and **`transcr`**. These implement tonal transposition and transcription. Both functions require scales as parameters. A scale is a sequence of notes, in which any pitch-class occurs no more than once (and which does not contain rests). Moreover, the tones of a scale should be ordered by pitch (increasing or decreasing) and and the ambitus is less than an octave.

Example: `[c d e f g a h]` is a scale in this sense (C major), as well as `[c g d2 a2]` (no conventional scale); `[c d e f g a h c2]` is, however, not a scale, as the pitch-class *c* occurs twice.

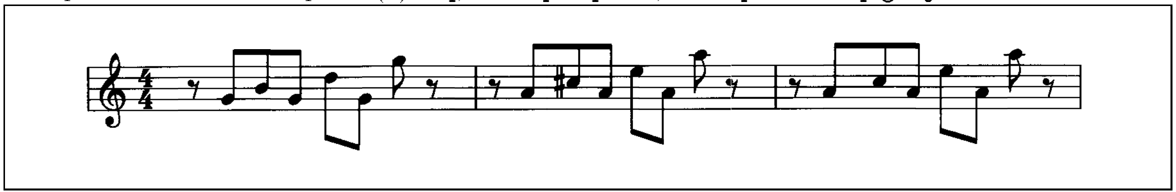
Tonal transposition is realised by transposing the pitches of the given sequence along the given scale by the given number of steps, where the scale is extended, if necessary, in both directions. Pitches not occurring in the scale are problematic; these notes are transformed into rests of the respective duration in the current realisation. Chromatic transposition (`transp`) can be understood as special case of tonal transposition, using the chromatic scale `[c c# d d#` . . . `a# h]` as scale.

Applying the transcription operations, the pitches of a given sequence are translated from one scale into another. This is done by first determining the position of each note's pitch with respect to the primary scale; it is then replaced by the pitch at the corresponding position of the destination scale. Again, the scales are extended, if necessary, in both directions. Pitches not included in the source scale remain unchanged. Tonal transposition is a special case of transcription, where the destination scale is a shifted version of the source scale.

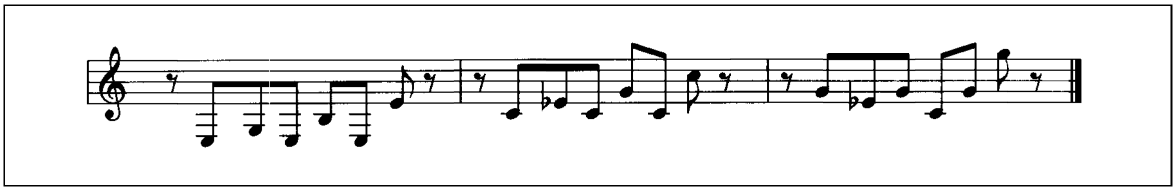
Extended transposition functions:

`transpTonal(`seq_1`,` seq_2`,` *int*`)`	returns the sequence which is obtained by tonal transposition of sequence seq_1 by *int* steps wrt. scale seq_2 (see above).
`transcr(`seq_1`,` seq_2`,` seq_3`)`	returns the sequence which is obtained by transcription of sequence seq_1 from scale seq_2 into the scale seq_3 (see above).

transposition and transcription (1): `sq`, `transp(sq,+2)`, `transpTonal(sq,gMaj,+1)`



transposition and transcription (2):
`transpTonal(sq,gMaj, -9)`, `transcr(sq, gMaj, cMin)`, `transcr(sq, gMaj, invert(gMaj))`



Example: Extended transposition functions

```
>sq := [_/8 g h g d2 g1 g2]
>gMaj := [g/4 a h c2 d e f#]
>cMin := [c/4 d e& f g a& h&]
>transp(sq,+2)
Result= [ I=80 M=100 T=0 _/8 a1/8 c#2/8 a1/8 e2/8 a1/8 a2/8 ]
>transpTonal(sq, gMaj, +1)
Result= [ I=80 M=100 T=0 _/8 a1/8 c2/8 a1/8 e2/8 a1/8 a2/8 ]
>transpTonal(sq, gMaj, -9)
Result= [ I=80 M=100 T=0 _/8 e0/8 g0/8 e0/8 h0/8 e0/8 e1/8 ]
>transcr(sq, gMaj, cMin)
Result= [ I=80 M=100 T=0 _/8 c1/8 e&1/8 c1/8 g1/8 c1/8 c2/8 ]
>transcr(sq, gMaj, invert(gMaj))
Result= [ I=80 M=100 T=0 _/8 g1/8 e&1/8 g1/8 c1/8 g1/8 g2/8 ]
>
```

The following functions are used for changing the alteration of given sequences.

Alteration functions:

`alter`(*seq*, *int*)	returns the sequence obtained from *seq* by setting the alteration of each note to *int*; where positive *int* values represent a corresponding number of sharps, while negative values represent flats. If *int* equals 0, all pitches are represented without alterations using chromatic pitch-classes when necessary.
`alterTonal`(*seq*, *int*)	returns an altered version of *seq* like `alter`, only here chromatic pitch-classes are never used but each pitch is represented by diatonic pitch-classes with, whenever possible, alteration level *int* (see examples below).

These functions do not affect the overall sounding pitches of a sequence, but merely change alteration levels in an enharmonic way.

Note:

- The system setting `noteNames` affects the representation of note names within sequences and segments. To actually see the internally represented alteration information, in particular, if chromatic pitch-classes may occur (whether explicitly specified or as an effect of `alter`),`noteNames` has to be set to `chromatic`.
- Besides applying alterations, `alterTonal` internally transforms notes with chromatic pitch-calsses into a corresponding diatonic representation. This can be used, for example, to remove unwanted chromatic pitch-classes arising from the transposition of scales.

Example: Alteration functions

```
>set("noteNames")
noteNames=diatonic
>alter([c d e],2)
Result= SEGM([ I=80 M=100 T=0 h#0/4 c##1/4 d##1/4 ])
>set("noteNames","chromatic")
>alter([c d e],2)
Result= SEGM([ I=80 M=100 T=0 ais##0/4 c##1/4 d##1/4 ])
>alterTonal({[c d e f#]},-1)
Result= SEGM([ I=80 M=100 T=0 c1/4 d1/4 f&1/4 g&1/4 ])
>set("noteNames","chromatic")
>alterTonal([cis],1)
Result= [ I=80 M=100 T=0 c#1/4 ]
>alterTonal([cis],0)
Result= [ I=80 M=100 T=0 c#1/4 ]
> >transp([c d e f g a h],1)
Result= [ I=80 M=100 T=0 cis1/4 dis1/4 f1/4 fis1/4 gis1/4 ais1/4 c2/4 ]
>set("noteNames","diatonic")
>transp([c d e f g a h],1)
Result= [ I=80 M=100 T=0 c#1/4 e&1/4 f1/4 f#1/4 g#1/4 h&1/4 c2/4 ]
>alterTonal(transp([c d e f g a h],1),-1)
Result= [ I=80 M=100 T=0 d&1/4 e&1/4 f1/4 g&1/4 a&1/4 h&1/4 c2/4 ]
>
```

The function `fragm(`seq`,`$real_1$`,`$real_2$`)` ist used to extract temporal sections from a piece. It returns the fragment of sequence *seq* starting at relative time position $real_1$ and having a relative duration of $real_2$. If the value for $real_1$ is less than 0 or greater than the relative duration of *seq*, the empty sequence is returned. In case of a negative relative duration $real_2$ the time position $real_1$ indicates the right margin (i.e. the end) of the fragment, while the duration is $|real_2|$. If the specified fragment exeeds the right (or left) margin of *seq*, it will be shortened respectively (it will not be filled up with rests!).

Example: Extraction of temporal fragments

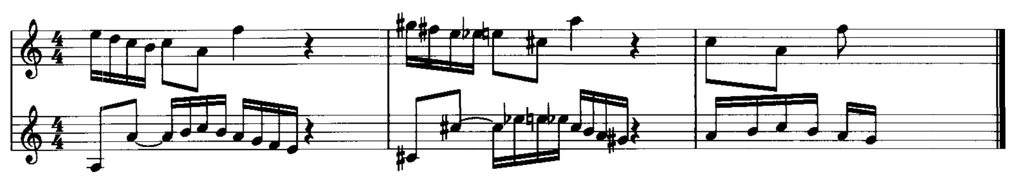
```
>sq := [_/8 g h g d2 g1 g2]
>fragm(sq, 5/16, 2/8)
Result= [ I=80 M=100 T=0 h1/16 g1/8 d2/16 ]
>fragm(sq, 2/4, 8/4)
Result= [ I=80 M=100 T=0 d2/8 g1/8 g2/8 ]
>fragm(sq, 2/4, -8/4)
Result= [ I=80 M=100 T=0 _/8 g1/8 h1/8 g1/8 ]
>
```

All variator functions on sequences that have been described above can also be applied to segments, in this case the corresponding operation is applied to each voice.

Example: Variator functions on segments

```
>sg := {[e2/16 d c h1 c2/8 a1 f2/4], [a0/8 a1*3/16 h/16 c2 h1 a g f e]}
>transp(sg, +4)
Result= SEGM([ I=80 M=100 T=0 g#2/16 f#2/16 e2/16 d#2/16 e2/8 c#2/8 a2/4 ], [ I=80
M=100 T=0 c#1/8 c#2*3/16 d#2/16 e2/16 d#2/16 c#2/16 h1/16 a1/16 g#1/16 ])
>fragm(sg, 1/4, 3/8)
Result= SEGM([ I=80 M=100 T=0 c2/8 a1/8 f2/8 ], [ I=80 M=100 T=0 a1/16 h1/16 c2/16
h1/16 a1/16 g1/16 ])
>
```

Variation of a segment: `sg`, `transp(sg,+4)`, `fragm(sg, 1/4, 3/8)`



5.7 Musical combinator functions

Salieri provides two functions for combining sequences: `comb` und `merge`. These functions are defined in the following way:

Combinator Functions:

`comb(`seq_1`,` seq_2`)`	returns the sequence obtained by combining the pitch sequence of seq_1 with the duration sequence of seq_2; the shorter sequence is repeated, if necessary; the secondary parameters are taken from seq_1.
`merge(`seq_1`,` seq_2`)`	returns the sequence obtained by concatenating the sequence seq_1 and seq_2 and merging the last tone t_1 of seq_1 and the first tone t_2 of seq_2; this is realised by extending the duration of t_1 by the duration of t_2 and deleting t_2 (see example below).

The function **comb** can be used to apply a recurrent rhythmic pattern to a given melody. In a way, the **merge** operation is a counterpart to **fragm** (which in general may split notes when extracting fragments); when applying **merge**, the relative durations of the sequences are added, but one pitch (namely the one of t_2, see above) is lost.

Example: Combinator functions

```
>hMaj := [h/4 c#2 d# e f# g# a#]
>comb(hMaj, [_/8 _/8 _/4])
Result= [ I=80 M=100 T=0 h1/8 c#2/8 d#2/4 e2/8 f#2/8 g#2/4 a#2/8 ]
>comb([c f], [_/8 _*3/8 _/8 _ _ _ _ _/4])
Result= [ I=80 M=100 T=0 c1/8 f1*3/8 c1/8 f1/8 c1/8 f1/8 c1/8 f1/4 ]
>[h/4 c#2 d#]+[e/4 d# c#]
Result= [ I=80 M=100 T=0 h1/4 c#2/4 d#2/4 e1/4 d#1/4 c#1/4 ]
>merge([h/4 c#2 d#], [e/4 d# c#])
Result= [ I=80 M=100 T=0 h1/4 c#2/4 d#2/2 d#1/4 c#1/4 ]
> >gMaj := [g/4 a h c2 d e f#]
>cMin := [c/4 d e& f g a& h&]
>transp(sq,+2)
Result= [ I=80 M=100 T=0 _/8 a1/8 c#2/8 a1/8 e2/8 a1/8 a2/8 ]
>transpTonal(sq, gMaj, +1)
Result= [
```

6 User-defined functions

The user of the SALIERI System is offered the possibility of defining new functions and thus expanding the system according to his requirements. Functions are generally treated as objects, and thus they are basically treated exactly as the objects of other types (integers, sequences etc.). In general there is a distinction between functions and macros depending on the handling of the parameters. At the end of this section the internal (predefined) functions, which are also realised as an object type are discussed briefly.

6.1 Function definition

Functions are defined by a statement of the form "*name* := FUNC(*int*, *body*);" where the name of the function *name* is a valid object name, *int* is the number of parameters, and the function body *body* a sequence of instructions which is delimited by single quotes `'...'`.

Comment: The function definition is really an assignment (cf. section 3.1), with **FUNC** being a constructor for functional objects.

Within the function body the parameters are accessed as `$1`, `$2` etc. according to their position in the parameter list of the function call. Conceptually, each function has its own (local) workspace. It contains all objects of the global workspace (as copies!) and the function parameters during the execution of the function, and it is automatically deleted when the function is terminated. Consequently, if an object is created within a function, it will only exist until the respective function call is completed. Therefore, while objects of the global workspace can be accessed within functions, they cannot be altered or deleted permanently.

The function **return** is used to terminate a function and to return a result. By means of the instruction **return(***expr***)** the execution of the function is terminated at this positon (i.e. subsequent statements in the function body will not be executed!), with the value of the expression *expr* being returned as the function result. The instruction **return()** has the same effect, except that in this case the function does not return a result (therefore it is rather to be considered as a procedure or command).

Example: Function definition

```
scaleMaj := FUNC(1,
  (* EFFECT: returns the major scale based on a given base note
     CALL: scaleMaj(<note>)
     EG: scaleMaj([c]) -> [c d e f g a h] *)
  'return ($1 + transp($1,2) + transp($1,4) + transp($1,5)
          + transp($1,7) + transp($1,9) + transp($1,11));
  ');
```

Note:

- Function definitions should generally be written in program files; in this way they can be easily modified.
- Functions are ordinary objects, in particular they can occur in assignments or as parameters of functions.
- The syntactical correctness of function bodies is generally not tested until the corresponding function is executed (and not when it is defined); also refer to section 11).
- Functions can be recursively defined, i.e. they can call themselves within their function body. The depth of recursion is limited by the available memory. ATTENTION: Too deep or endless recursion can result in system crashes (loss of data!).
- There is no automatic type checking for function parameters. Thus it is possible to realise functions which can be applied to objects of different types (polymorphism). The user has to take care that no type conflicts will arise when user-defined functions are executed. It is highly recommended to note the admissible types of parameters for functions in a comment or to carry out an explicit type checking (cf. example).
- Although function parameters such as `$1` are ordinary objects of the local workspace, they should not be overwritten or deleted. Furtheron, for more complex functions it is helpful to assign the parameter values to local objects with suggestive names at the beginning of the function body.
- Basically, arbitrary objects can be returned as result. However, attention has to be paid if a function `f1` returns a functional object `f2` as result. In this case local objects of `f1` are not allowed to occur in the body of `f2`, as these objects are only defined while `f1` is being executed.
- Function definitions can be nested, i.e. local functions can be defined within a function body. However, for clarity's sake this should be avoided when possible.

Comment: The equality operators "'`=`"' and "'`!=`"' can be applied to functions, with two functions being equal if they have the same number of parameters as well as function bodies which are syntactically identical.

6.2 Function call

User-defined functions are called like internal functions using the syntax *name*`(`par_1`, ..., `par_n`)`. Here, *name* is the name of the function and par_1 etc. are the parameter value. Any expression can occur in place of a parameter value. There are also parameter-less functions; in this case the function call syntax is *name*`()`.

If the corresponding function returns a result, the function call is an expression which evaluates to the function result. If no result is returned, the function call is a command and can consequently not be used as the right hand side of an assignment or as a functional parameter.

Example: Function call

```
>scaleMaj([d#/8])
Result= [ I=80 M=100 T=0 d#1/8 f1/8 g1/8 g#1/8 a#1/8 c2/8 d2/8 ]
>
```

Comment: An alternative form of the function call uses the internal function `apply`, where `apply(`*name*`,` par_1`, ..., `par_n`)` exactly corresponds to the standard function call *name*`(`par_1`, ..., `par_n`)`. For clarity's sake, the use of `apply` should be generally avoided.

6.3 Macros

Macros are special functions that are defined by a statement of the form *name* `:= MACRO(`*int*`, `*body*`)`. The only difference to common functions is that objects defined within macros are generally not local. In macros, the names of local objects have to begin with $. This notation is also allowed for functions, it does, however, not result in an additional functionality in this case. All object names without a leading $ occuring in a macro body, consequently refer to the global workspace and are therefore to be treated with caution (unwanted side effects by overwriting objects).

Macros are especially suited for the realisation of system commands, such as, e.g., displaying or deleting of certain groups of objects, or initialising certain objects and data structures.

Example: Macro definition

```
setMyScales := MACRO(1,
  (* EFFECT: sets the variables myT,myD,myS to the major scales
             of the tonic, the dominant and the subdominant
             w.r.t.  a given keynote
     CALL: setMyScales(<note>)
     EG: setMyScales([c]) *)
  `myT := scaleMaj($1);
  myD := transp(myT,+7);
  myS := transp(myT,+5);
  ');
```

Example: Macro call

```
>setMyScales([g]);
>writeAll("my*");
myD = [ I=80 M=100 T=0 d2/4 e2/4 f#2/4 g2/4 a2/4 h2/4 c#3/4 ] : sequence
myS = [ I=80 M=100 T=0 c2/4 d2/4 e2/4 f2/4 g2/4 a2/4 h2/4 ] : sequence
myT = [ I=80 M=100 T=0 g1/4 a1/4 h1/4 c2/4 d2/4 e2/4 f#2/4 ] : sequence
>
```

The equality operators ”‘`=`”’ and ”‘`!=`”’ can be applied to macros, with two macros being equal if they have the same number of parameters as well as macro bodies which are syntactically identical. Macros and functions can be compared and are basically considered to be unequal.

6.4 Functions with a variable number of parameters

It is often useful when a user-defined function can be called with a variable number of parameters. This can be realised in SALIERI by specifying the number of parameters as −1 when the function is defined. In this case, when the function is called, all parameters are combined into a list which can be accessed as `$1` within the function body. Such a function can be called using an arbitrary number of parameters (even without parameters, then the parameter list `$1` is empty).

Example: Definition of a function with variable number of parameters

```
demoVarPar := FUNC(-1,
  (* CALL: demoVarPar(<any_1>, ..., <any_n>)
     EG: demoVarPar(1,2,3, [c d e], [c# d# f]) *)
  `parlist := $1;
  write("Number of parameters: ", #parlist);
  write("Parameter list: ", parlist);
  ');
```

Example: Call of a function with variable number of parameters

```
>demoVarPar(1,2,3, [c d e], [c# d# f]);
Number of parameters: 5
Parameter list: LIST(1,2,3,[ I=80 M=100 T=0 c1/4 d1/4 e1/4 ],[ I=80 M=100 T=0 c#1/4 d#1/4
f1/4 ])
>
```

Comment: Of course, macros with a variable number of parameters can be realised in the same way.

6.5 Internal functions

Like user-defined functions or macros, internal functions are regarded as objects. In particular, they can occur in assignments or as functional parameters.

This can be used, e.g., for "‘renaming"’ system commands, by copying the corresponding internal function to an object with a different name.

Example: Internal functions

```
>ls := statAll;
>ls
Result= IFUNC("statAll")
> ls("*");
Objects matching *:

demoVarPar: function (size=109)
ls : internal function (size=22)
>
```

Note:

Internal functions cannot be redefined or deleted. When an arbitrary object is accessed, the system first looks for an internal functional object with the corresponding name; only if such an object is not found, the object is searched in the workspace. Thus, although it is possible to define, for example, an object named `exit`, the (predefined) internal function will be executed when `exit` is called.

Comments: The equality operators "'`=`"' and "'`!=`"' can be applied to internal functions. Two internal functional objects are equal if they denote the same internal function. Internal functional objects can be compared with macros and functions and are generally considered to be unequal to these.

7 Control structures

Common control structures such as conditionals or loops are generally required for specifying the control flow within user-defined functions; in SALIERI, these are realised as functions.

At first we introduce the predefined control functions `if`, `while`, `loop` and `loopn`; then we discuss briefly the realisation of new control structures within the system.

7.1 Blocks

Blocks are sequences of SALIERI instructions delimited by single quotes `‘...’`. Strictly speaking, a block is a parameter-less macro, therefore the notations `‘...’` and `MACRO(0,‘...’)` are basically equivalent.

There is a distinction between statement blocks and expression blocks: the former contain instructions only, whereas the latter include an expression (or a `return` instruction) and thus their evaluation yields a result value.

Example: `‘x := x+1’` and `‘x := y; y := f(x); write(y)’` are statement blocks; `‘2 > 1’` and `‘x := x+1; return(x > 5)’` are expression blocks (the former of which evaluates to `TRUE`). `‘’` is also a valid statement block (the empty block), which has no effect nor result when executed. `‘x < 1; x > 5’` is not a valid block, as it includes two *expressions* seperated by "';"' (which is generally not allowed).

Comment: Blocks are, for example, required for realising control functions such as loops, as in SALIERI function parameters are generally evaluated before a function is called (call-by-value). Combining a sequence of statements into a block prevents these from being executed immediately when the appear as parameter of a control function. To execute a block, usually the function `apply` is used (compare 6.2).

7.2 Conditionals

The control function `if` realises the conditional execution of blocks. It is generally used as a statement in one of the forms `if(`*cond-block*`, `*if-block*`)` or `if(`*cond-block*`, `*if-block*`, `*else-block*`)`.

Here, *cond-block* is a block which yields a truth value when evaluated; *if-block* is a statement block which is only executed if the evaluation of *cond-block* results in the value `TRUE`. There is no need to specify the statement block *else-block*; in case it is available, it will only be executed, if the evaluation of *cond-block* does not result in the value `TRUE`.

Example: Conditional

```
>sq := [h e&];
>if('#sq > 0 and #sq < 8', 'sq := sq+transp(sq,2)');
>write(sq);
[ I=80 M=100 T=0 h1/4 d#1/4 c#2/4 f1/4 ]
>if('rnd() <= 0.5', 'sq := sq+transp(sq,2)', 'sq := retro(sq)');
>write(sq);
[ I=80 M=100 T=0 h1/4 d#1/4 c#2/4 f1/4 c#2/4 f1/4 d#2/4 g1/4 ]
>
```

Comment: The parameters *if-block* and *else-block* of the `if` function can also be expression blocks (however, both of them have to be expression blocks!); in this case the entire function is an expression rather than a statement. This feature should, however, only be used by experienced users.

By means of `if` it is also possible to realise multiple conditioning without excessive nesting (similar to the *case* statement in the C programming language). This is realised by using a statement of the form `if(`*cond-block*$_1$, *if-block*$_1$, *cond-block*$_2$, *if-block*$_2$, ... , *else-block*`)`, where *else-block* is optional as for the simple `if`.

Example: Multiple conditioning

```
>case:=FUNC(1,'if('$1<0','return("-")','$1=0','return("null")','return("+")')')
>case(88)
Result= "+"
>case(-7.7)
Result= "-"
>case(0)
Result= "null"
>
```

7.3 Loops

Salieri provides three builtin loop functions: `while`, `loop` und `loopn`. These control functions are used in the following way:

Syntax of the loop functions:

```
while(cond-block, body-block)
loop(init-block, cond-block, iter-block, body-block)
loopn(int, body-block)
```

In the above, *cond-block* is an expression block with result type `"bool"`, and *body-block* is a statement block containing the loop body.

When a `while` loop is executed, the loop body *body-block* is executed as long as the evaluation of *cond-block* returns `TRUE`.

Example: `while` loop

```
>sq := [h e&];
>while('#sq > 0 and #sq < 8', 'sq := sq+transp(sq,2)');
>write(sq);
[ I=80 M=100 T=0 h1/4 d#1/4 c#2/4 f1/4 c#2/4 f1/4 d#2/4 g1/4 ]
>
```

`loop` is the most general loop function (it corresponds to the for-loop in the programming language C). If a `loop` statement is executed, the statement block *init-block* is executed (once) at first, which generally contains the initialisation of the loop (such as initialising counter variables). Subsequently the loop body *body-block* and the statement block *iter-block* are executed (in that order), as long as *cond-block* evaluates to `TRUE`; *iter-block* should contain statements for controlling the loop (such as increasing counter variables).

Example: `loop` loop

```
>sq := [h& a c h];
>sq2 := [];
>loop('i := 1', 'i <= #sq', 'i := i+2','sq2 := sq2 + infix(sq,i,1)');
>write(sq2);
[ I=80 M=100 T=0 a#1/4 c1/4 ]
>
```

The `loopn` function is used to execute the loop body *body-block* a fixed number of *int* times, if *int* is an integer (or a corresponding expression) which is greater than zero (otherwise nothing is done).

Example: `loopn` loop

```
>sq := [c];
>sq2 := [];
>loopn(10, 'sq2 := sq2 + transp(sq,rndUniform(-5,+5))');
>write(sq2);
[ I=80 M=100 T=0 c1/4 d#1/4 a#0/4 c1/4 c1/4 c1/4 d#1/4 e1/4 d#1/4 f1/4 ]
>
```

Note:

- For clarity's sake, *cond-block* should generally contain a boolean expression (and should not cause any side-effects).
- The functionality of the `while` and `loopn` loop can also be achieved by means of the `loop` instruction. However, in general `loopn` is executed considerably faster than an equivalent formulation using `while`; the same applies to `while` with respect to `loop`. Nevertheless the clearest and most comprehensive solution (if possible) should be preferred when chosing a specific formulation of a loop.

7.4 User-defined control structures

As in SALIERI functions (and consequently blocks, as well) can be occur as function parameters, additional control structures can be realised in a simple way as user-definied functions. In this case the blocks passed as parameters are evaluated using the function `apply` (in the case of expression blocks) or executed (in the case of statement blocks).

The following example realises an `until` loop, the body of which is executed repeatedly (at least once), until a given termination condition is satisfied.

Example: Definition of an `until` loop

```
doUntil := MACRO(2,
  (* EFFECT: executes the statement block $1=<block> as often as the expression-
             block $2=<cond> results in TRUE
     CALL: doUntil(<block>, <cond>)
     EG: doUntil('i := i+1', 'i > 10') *)
  'apply($1);
  while ('not (apply($2) = TRUE)','apply($1)');
  ');
```

Example: Use of the `until` loop

```
>i := 1;
>doUntil('write(i); i:=i+2', 'i > 4');
1
3
>i
Result= 5
>
```

The final example shows how to realise a conditional expression which yields different results depending on a boolean value.

Example: Definition of a conditional expression

```
cond := MACRO(3,
  (* EFFECT: returns the value $2=<if-expr>, if the evaluation of the boolean
             expression $1=<cond-expr> results in TRUE, value $3=<else-expr> otherwise
     CALL: cond(<cond-expr>, <if-expr>, <else-expr>)
     Ex:  cond(n > 0, n, -n) *)
  'if('$1=TRUE',
     'return($2)',
     'return($3)'
     );
  ');
```

Example: Using the conditional expression

```
>sq := [h e&];
>sq := cond(rnd() <= 0.5, sq+transp(sq,2), retro(sq));
>sq
[ I=80 M=100 T=0 h1/4 d#1/4 c#2/4 f1/4 ]
>
```

8 Data structures

Besides the simple and musical data types, SALIERI provides the possibility of forming (heterogenous) lists over arbitrary objects. Such a list can contain objects of different types at the same time; based on this concept, more complex structures can be realised (Note: a list can also contain other lists as elements, which can be used to represent general tree structures).

In the following, we first introduce the object type `list` and its basic operations. Subsequently we demonstrate the realization of other data structures using list objects.

8.1 Lists

Lists are created using expressions of the form `LIST(`obj_1`, ..., `obj_n`)` where obj_1 etc. are arbitrary objects (or expressions). As a shorthand notation the constructor `LIST` can be omitted for lists with more than one element. Also, the empty list can be denoted by `()`.

Example:
`LIST(1,2,3)`, `(1,2,3)`, `LIST([c/4 e g],0.25,"test")` and `LIST( LIST(1/4,TRUE), FUNC(1,'return($1*2)'))` are valid lists. `LIST()` or `()` is a valid list (the empty list) in particular.

Besides the serial operations `+`, `infix`, `ins`, `del`, `length` and `posNext` (refer to Section 4.4), there are five other basic operations on lists:

List Functions:

`LISTINIT(`*int*`, `*obj*`)`	returns the list containing *int* times the object *obj*.
`consList(`*obj*`, `*list*`)`	returns the list obtained by inserting the object *obj* into the list *list* at its head (before position 1).
`getList(`*list*`, `*int*`)`	returns an element at position *int* of the list *list* (positions are labelled 1,2,etc.).
`putList(`*list*`, `*obj*`, `*int*`)`	returns the list obtained from *list* by replacing the element at position *int* by object *obj*.
`inList(`*list*`, `*obj*`)` $\rightsquigarrow$ *bool*	returns `TRUE` if the list *list* contains the object *obj* as element, otherwise `FALSE`.

Notations: `#`*list* can be written as a shorthand for `length(`*list*`)`, with the operator "'`#`"' having precedence over "'`*`"' and "'`/`"'. *list*`@`*int* can be as a shorthand for `getList(`*list*`, `*int*`)`; with the operator "'`@`"' having precedence over any other operator. Also, *list*`@`*int* `:=` *obj* can be used instead of *list* `:= putList(`*list*`,`*obj*`,`*int*`)` if *list* is the name of a list object.

Example: List operations

```
>LISTINIT(5,"a") Result= LIST("a","a","a","a","a") >ls := LIST([c],[e],[g],[h],[c2])
>getList(ls,#ls)
Result= [ I=80 M=100 T=0 c2/4 ]
>inList(ls,[e])
Result= TRUE
>putList(ls,[e&], posNext(ls,LIST([e]),1))
Result= LIST([ I=80 M=100 T=0 c1/4 ],[ I=80 M=100 T=0 d#1/4 ],[ I=80 M=100 T=0 g1/4
],[ I=80 M=100 T=0 h1/4 ],[ I=80 M=100 T=0 c2/4 ])
>infix(ls,1,3)
Result= LIST([ I=80 M=100 T=0 c1/4 ],[ I=80 M=100 T=0 e1/4 ],[ I=80 M=100 T=0 g1/4 ])
>consList([c0], ls)
Result= LIST([ I=80 M=100 T=0 c0/4 ],[ I=80 M=100 T=0 c1/4 ],[ I=80 M=100 T=0 e1/4 ],[
I=80 M=100 T=0 g1/4 ],[ I=80 M=100 T=0 h1/4 ],[ I=80 M=100 T=0 c2/4 ])
>sq := [];
>loop('i := 1', 'i <= #ls', 'i := i+1','sq := sq + ls@i');
>sq
Result= [ I=80 M=100 T=0 c1/4 e1/4 g1/4 h1/4 c2/4 ]
>
```

The equality operators ”‘`=`”’ und ”‘`!=`”’ can be applied to lists, with two lists being equal if they have the same length and all elements are equal.

Lists can basically reach a length of up to 32767 elements.

8.2 User-defined data structures

On the basis of the list concept virtually any other data structure can be realised. The implementation using list structures is achieved by defining a set of access functions. As the lists which realise the new data structures generally have to satisfy specific properties, the access functions should ensure that these properties are preserved.

In the following example the implementation of a dynamic record-structure is demonstrated. A record is a set of data which is stored in individual fields that are accessed by field names. Each record object is realised as a list containing alternately a field name and the corresponding field contents. Following operations are provided for this data strucure: creating an empty record, overwriting a field, retrieving the contents of a field, copying and deleting entire records. Deleting and copying can be performed as for any other SALIERI object; the remaining operations are realised as functions and combined into a library.

Example: Library `RECORD.SAL`

```
(* SALIERI 2.0
   LIBRARY record.sal
   CONTENTS: The implementation of the abstract data type "record"
   FUNCTIONS: newRec, putRec, getRec *)

(* record = LIST(<str_1>, <any_1>, ..., <str_n>, <any_n>) *)

newRec := FUNC(-1,
```

```
  (* EFFECT: creates record with the field names <str_1>, ..., <str_n>
     CALL:  newRec(<str_1>, ..., <str_n>) -> record
     EX:     newRec("title","composer","date") *)
  `names := $1;
  res := LIST();
  i := 1;
  while(`i <= #names',
    `res := res + LIST(names@i, "");
    i := i+1;
    ');
  return(res);
  ');

putRec := FUNC(3,
  (* EFFECT: yields <record>, after the contents of the field <str>
              was set to <any>
     CALL:  putRec(<record>,<str>,<any>) -> <record>
     EX:     putRec(myrec,"title","Pr\"oludium in C") *)
  `rec := $1; fieldname := $2; fieldval := $3;
  i := 1;
  while (`(i <= #rec) and (rec@i != fieldname)',
    `i := i+2;
    ');
  if (`i <= #rec', `rec := putList(rec,$3,i+1);');
    return(rec);
  ');

getRec := FUNC(2,
  (* EFFECT: yields the contents of the field <str> of <record>,
     CALL:  getRec(<record>,<str>) -> <any>
     EX:     getRec(myrec,"title") *)
  `rec := $1; fieldname := $2;
  i := 1;
  while (`(i <= #rec) and (rec@i != fieldname)',
    `i := i+2;
    ');
  if (`i <= #rec',
    `return(rec@(i+1))',
    `return("")'
    );
  ');
```

This library can be used in the following way:

Example: Usage of the library `RECORD.SAL`

```
>exec("record")
Executing record.sal ...
>emptyRec := newRec("title","composer","date");
>myRec := putRec(emptyRec,"title", "Präludium in C");
>myRec
Result= LIST("title","Präludium in C","composer","","date","")
>myRec := putRec(myRec,"composer", "J.S.Bach");
>myRec
Result=LIST("title","Pröludium in C","composer","J.S.Bach","date","")
>write(getRec(myRec,"title"));
Präludium in C
>write(getRec(myRec,"composer"));
J.S.Bach
>
```

9 Input/Output Functions

In order realise user interaction within SALIERI programs, the SALIERI language offers basic input and output operations as well as some functions for MIDI output of musical objects.

Input/output functions

`read(`*string*`)`	displays a prompt *string* and returns the value of the object typed in by the user subsequently.
`readStr(`*string*`)`	displays a prompt *string* and returns the character string typed in by the user subsequently.
`write(`obj_1`, `obj_2`, ..., `obj_n`)`	writes obj_1 to obj_n (without separating these by spaces) to the SALIERI system dialogue.
`saveAsMidi(`*string*`, `*segm*`, `int_1`, `int_2`)`	saves the segment *segm* as a MIDI file named *string*. Optionally the nominator and the denominator of the desired metre can be specified using int_1 and int_2.
`midiPlay(`*segm*`)`	triggers play-back of the segment *segm*.

Objects of any type can be read using the function `read`. In doing so, the SALIERI System interprets the input as an expression and returns the result of the evaluation. If the user input should not be automatically interpreted, but rather stored in string, the function `readStr` should be used instead.

Note:

- If the evaluation of the input is not intended, the function `readStr` has to be used.
- If one desires to put in a character string with the function `read`, it has to be delimited by double quotes. This is not required when using the fuction `readStr`.
- The function `write` puts out its parameters without any space in between.

The function `saveAsMidi` writes the given segment into a MIDI file, which can be played or further processed later on. For interactive operation, the SALIERI System also offers the possibility of playing back segments using the function `midiPlay`.

Note:

- By means of the implicit type conversion within SALIERI, `saveAsMidi` and `midiPlay` can be also applied to note sequences instead of segments.
- For `saveAsMidi` and `midiPlay`, the extension `.mid` is automatically added to the file name when no other extension is explicitly specified.

Example: Input/output functions

```
>read("Input: ")
Input: 12
Result= 12
>read("Input: ")
Input: 5<7
Result= TRUE
>read("Input: ")
Input: abcd
Error in command-line: Unknown identifier: abcd
(occurred while executing function "read")
>read("Input: ")
Input: "abcd"
Result= "abcd"
>readStr("Input: ")
Input: abcd
Result= "abcd"
>readStr("Input: ")
Input: 5<7
Result= "5<7"
>write("hello ", "world!", 12, 3)
hello world!123
>saveAsMidi("test", [c1/4 d/8 g d/4 e])
Creating MIDI-file test.mid ... Done
>saveAsMidi("test34", [c1/4 d/8 g d/4 e],3,4)
Creating MIDI-file test.mid ... Done
>midiPlay([{e1/4, c} {f1/8, d} {g1/8, c} {f1/4, d} {g1/4, e}])
Creating MIDI-file U:\SALIERI\_tmp0.mid ... Done
Creating MIDI-file U:\SALIERI\_tmp0.ply ... Done
>
```

Comment: The function `midiPlay` creates two MIDI files, where the second file (with the ending `.ply`) indicates to the external play-back mechanism that a MIDI file is ready to be played.

10 Additional System Commands

Besides the system commands introduced in the previous sections (compare Sections 3, 2) there are two additional commands available: `execDOS` und `dir`.

The `execDOS` commmand serves the purpose of executing DOS commands and can be used in two different ways: Using the form `execDOS()`, a DOS shell is executed which allows the execution of arbitrary DOS commands while the Salieri System remains resident in the background. In this case, the DOS command `EXIT` is used to terminate the DOS shell and to return to the Salieri System. The alternate variant `execDOS(`*string*`)` executes a single DOS command line *string* (which has to be specified as character string, cf. example below).

In addition the command `dir(`*maskStr*`)` is available; this command displays the names of all files in the current working directory which match the template *maskStr* (cf. section 3).

Example: Additional system commands

```
>execDOS()
Type EXIT to return to Salieri ...

Microsoft(R) MS-DOS(R), Version 6.22
(C)Copyright Microsoft Corp 1981-1994.

E:\SALIERI>exit

... and here you are back in SALIERI again.
>execDOS("cd");
E:\SALIERI\
>dir("sal*ri*")
Directory-entries matching SAL*RI*:

SALIERI.EXE
SALIERI.LOG
SALIERI.WSP
>
```

Note:

- In order to avoid loss of data in the case of a program or system failure, the workspace should be saved using `sync()` before executing any DOS commands or starting a DOS shell.
- In order to quit a DOS shell, the DOS command `EXIT` has to be used (rather than the SALIERI command `exit()`).

11 Error Handling

Time and again there will be misunderstandings between the user and the SALIERI System, which the latter one (presumptuously) interprets as errors made by the user — unfortunatle a quite common experience when working with computers.

Often, error conditions occur because a function or object name is mis-spelled or the number or types of the function parameters are not compatible with the specification of the function used.

Example: Incorrect function call

```
>writ("oops")
Error in command-line: Unknown function: writ
(occurred while executing function "writ")
>rndUniform()
Error in command-line: Wrong number of parameters
(occurred while executing function "rndUniform")
>infix("Salieri",3,2)+2
Error in command-line: Type mismatch
(occurred while executing function "+")
>
```

Another common source of errors is the incorrect usage of delimiters. In addition to parentheses, SALIERI uses other delimiters in various ways: double quotes for character strings, square brackets for note sequences, curly braces for note segments, single quotes for blocks, parentheses for expressions and within function calls.

Example: Incorrect parentheses

```
>(12+(3*2)))
Error in command-line: parse error
>'write()''
Error in line 1: "'" missing
>[c d e []
[Result= [I=80 M=100 T=0 c1/4 d1/4 e1/4]
>
```

If an error occurs in the specification of a user-defined function, it is often only encountered when the function is executed. The following example shows this for the frequent case of a missing semicolon:

Example: Example file `error.sal`

```
f := FUNC(0,
     (* Missing semicolon in line 4 *)
     'write("J.");
     write("S.")
     write("Bach");
');
```

Example: Missing semicolon in a user-defined function

```
>exec("error")
Executing error.sal ...
>f
Result= FUNC(0, `
            write("J.");
            write("S.")
            write("Bach");
            ')
>f()
Error in function "f", line 4: parse error
(occurred while executing function "f")
>
```

The missing of a semicolon is generally not detected until the next line, which makes the line number reported in the error message inaccurate. The line numbers reported for errors within user-defined function are relative to the function body (i.e. line 1 is the line in which the opening "'`"' is located).

Note:

As in certain situations system crashes can occur, it is advisable to save the contents of the workspace every now and then using the `sync` command.

Please report any errors in this manual or any bugs you may find using the SALIERI *System to* `hoos@informatik.th-darmstadt.de`*. We will also appreciate your suggestions and ideas. Your feedback will help us considerably in improving the* SALIERI *System.*

Index