**Technische Hochschule Darmstadt**
**Fachbereich Informatik**
**Institut für Theoretische Informatik**

# SALIERI v2.0

# Benutzerhandbuch

Holger H. Hoos und Thomas Helbich

April 1995

*Antonio Salieri (∗ Legnano 18.Aug.1750, † Wien 7.Mai 1825) war Zeitgenosse (und vermeintlicher „Gegenspieler") W.A.Mozarts in Wien, wo er unter anderem die Stellung des Hofkapellmeisters und (später) des Oberleiters des Konservatoriums innehatte. Salieri war ein bedeutender Lehrer seiner Zeit, er unterrichtete unter anderem L.v.Beethoven, F.Schubert, F.Liszt und S.Sechter (der seinerseits Lehrer A.Bruckners war). Der Schwerpunkt seines musikalischen Werkes lag im Bereich der Oper neapolitanischer Schule; er komponierte jedoch auch zahlreiche Werke der Instrumental- und Kirchenmusik. Seine Kompositionen werden heute, trotz ihrer musikalischen Qualitäten, nur sehr selten aufgeführt und sind weitgehend unbekannt.*

# Inhaltsverzeichnis

**1 Einleitung** **2**

**2 Installation und Aufruf** **3**

2.1 Systemanforderungen . . . . . 3

2.2 Systemdateien . . . . . 4

2.3 Umgebungsvariablen . . . . . 4

2.4 Aufruf und Beendigung . . . . . 4

**3 Grundsätzliches zur Bedienung** **5**

3.1 Kommandozeileneingabe . . . . . 5

3.2 SALIERI-Dateien . . . . . 6

**4 Workspace und Objekte** **7**

4.1 Objekte erzeugen und löschen . . . . . 8

4.2 Objekte inspizieren . . . . . 9

4.3 Speichern und Laden des Workspace . . . . . 10

4.4 Persistenz . . . . . 10

**5 Einfache Objekttypen** **10**

5.1 Wahrheitswerte . . . . . 10

5.2 Integer . . . . . 11

5.3 Reals . . . . . 12

5.4 Zeichenketten . . . . . 14

5.5 Vergleichsoperationen . . . . . 16

**6 Musikalische Objekttypen** **17**

6.1 Sequenzkonstanten . . . . . 17

6.2 Segmentkonstanten . . . . . 19

6.3 Basisoperationen auf Sequenzen . . . . . 20

6.4 Basisoperationen auf Segmenten . . . . . 22

6.5 Musikalische Deskriptorfunktionen . . . . . 23

6.6 Musikalische Variationsfunktionen . . . . . 25

6.7 Musikalische Kombinationsfunktionen . . . . . 30

**7 Benutzerdefinierte Funktionen** **31**

7.1 Funktionsdefinition . . . . . 31

7.2 Funktionsaufruf . . . . . 33

7.3 Makros . . . . . 33

7.4 Funktionen mit variabler Parameteranzahl . . . . . 34

7.5 Interne Funktionen . . . . . 35

**8 Kontrollstrukturen** **36**

8.1 Blöcke . . . . . 36

8.2 Verzweigung . . . . . 36

8.3 Schleifen . . . . . 37

8.4 Benutzerdefinierte Kontrollstrukturen . . . . . 39

**9 Datenstrukturen** **40**

9.1 Listen . . . . . 40

9.2 Realisierung weiterer Datenstrukturen . . . . . 41

**10 Ein- und Ausgabefunktionen** **43**

**11 Sonstige Systemkommandos** **45**

**12 Fehlerbehandlung** **46**

**Index** **49**

# Konventionen und Schreibweisen

In diesem Dokument werden folgende Informationen eingerahmt:

- Wichtige Punkte, welche der Benutzer beachten sollte, wie etwa

  **Beachte:**

  | |
  |---|
  | Für die korrekte Funktion dieser Software wird keinerlei Gewährleistung übernommen. Für Datenverluste oder sonstige Schäden, die im Zusammenhang mit dem Einsatz oder Betrieb entstehen, ist allein der Benutzer verantwortlich. |

- In Beispielsitzungen und Beispieldateien werden die Eingabe des Benutzer in „Typewriter“ und die Systemantwort kursiv gedruckt, wie z.B.

  **Beispiel:** Beispielsitzung

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

- Syntax von Aufrufen und SALIERI-Operationen, wie in

  **Variation der Nebenparameter:**

  | | |
  |---|---|
  | `cresc(`*seq*`,` $int_1$`,` $int_2$`)` | liefert die Sequenz, die aus *seq* ensteht, wenn diese mit einem linearen Intensitätsverlauf von $int_1$ nach $int_2$ versehen wird. |
  | `accel(`*seq*`,` $int_1$`,` $int_2$`)` | liefert die Sequenz, die aus *seq* ensteht, wenn diese mit einem linearen Metrumsverlauf von $int_1$ nach $int_2$ versehen wird. |

Des weiteren werden folgende hervorgehobenen Schriftarten verwendet:

- Die Systembezeichnung SALIERI erscheint in Kapitälchen.
- SALIERI-Operationen und Ausdrücke werden in „Typewriter“ angegeben, z.B. `write()` oder `[c1/4 d/8 g d/4 e]`.
- Die Parameter der SALIERI-Operationen sind kursiv gesetzt, z.B. `length(`*string*`)`.
- Betriebssystemnamen, also Dateinamen, Pfade und Umgebungsvariablen, werden in „Typewriter“ gesetzt, z.B. `SALIERI.EXE` oder `LIBS\*.SAL`.

# 1 Einleitung

Das SALIERI-System ist ein experimentelles System zur strukturorientierten Erzeugung und Bearbeitung von Musik. Das System liegt zur Zeit in der Version 2.0 vor und läuft auf PCs unter MS-DOS.

SALIERI besteht im wesentlichen aus zwei Komponenten: dem Interpreter der SALIERI-Sprache und einer interaktiven Entwicklungsumgebung. Die Bedienung des Systems erfolgt über eine zeilenorientierte Benutzerschnittstelle. Musikalische Objekte können als Dateien im MIDI-Format gespeichert werden; damit ist eine Anbindung an Wiedergabe- und Notensatzsysteme prinzipiell gegeben. Beim Einsatz von SALIERI innerhalb einer Multitasking-Umgebung können Tonsequenzen durch Ansteuerung eines speziellen Hintergrundprogrammes (Wiedergabe-Dämon) direkt aus dem SALIERI-System heraus abgespielt werden. Auf diese Weise wird ein interaktives Bearbeiten musikalischer Strukturen ermöglicht.

Die SALIERI-Sprache ist eine universelle, im wesentlichen funktionale Sprache, die insbesondere die strukturorientierte Darstellung ein- und mehrstimmigen musikalischen Materials unterstützt. Zur Darstellung von musikalischen Objekten wird ein einfaches, wahrnehmungsbezogenes Tonmodell verwendet, das wesentliche Teile der konventionellen Musiknotation in einer symbolischen Schreibweise umfaßt.

Die Sprache verfügt neben den Typen für musikalische Objekte über Basistypen für Wahrheitswerte, Zahlen und Zeichenketten. Zur Realisierung komplexerer Datenstrukturen können Listen über Objekten beliebiger, auch verschiedener Typen gebildet werden.

SALIERI verfügt über ca. 100 vordefinierte Funktionen; diese realisieren Operationen auf den verschiedenen Datentypen, Kontrollstrukturen und Systemkommandos. Für musikalische Objekte werden zahlreiche Variations- und Kombinationsoperationen angeboten. Die SALIERI-Sprache erlaubt die Definition von Funktionen und ermöglicht so die Anpassung des Systems an die Bedürfnisse des Benutzers. Insbesondere können auf diese Weise auf der Grundlage der vom System zur Verfügung gestellten vordefinierten Variations- und Kombinationsoperationen für musikalische Objekte beliebige neue Funktionen zur Manipulation musikalischen Materials realisiert werden.

Das SALIERI-System entstand in erster Linie in der Absicht, grundlegende Prinzipien und Aspekte der strukturorientierten Darstellung und Bearbeitung von Musik in Form eines interaktiven Systems zu realisieren. Die vorliegende experimentelle Realisierung läßt sicherlich viele Wünsche offen; die Benutzeroberfläche ist sehr spartanisch und die Reaktionen des Systems auf Fehler bei der Bedienung für ernsthafte Anwendungen stark verbesserungsbedürftig.

Eine Weiterentwicklung des Systems (vor allem bezüglich der Benutzeroberfläche) ist geplant, wird aber im einzelnen stark von den Erfahrungen beim Einsatz abhängen.

**Beachte:**

Da es sich bei der vorliegenden Software um ein experimentelles System handelt, kann es leicht zu Abstürzen oder Datenverlust kommen. Es empfiehlt sich daher dringend,

- die Benutzerdaten (Workspace) häufig zu sichern (mithilfe des `sync()`-Befehls, siehe Abschnitt 4.4);
- beim Einsatz unter Multitasking-Systemen (wie MS-Windows) die Benutzerdaten aller aktiver Programme vor Aufruf von SALIERI zu sichern;
- externe Sicherheitskopien aller wichtigen Daten, die sich auf dem entsprechenden Rechner befinden, anzulegen (z.B. auf Disketten).

**Für die korrekte Funktion dieser Software wird keinerlei Gewährleistung übernommen. Für Datenverluste oder sonstige Schäden, die im Zusammenhang mit dem Einsatz oder Betrieb entstehen, ist allein der Benutzer verantwortlich.**

# 2 Installation und Aufruf

## 2.1 Systemanforderungen

SALIERI ist für den Einsatz auf einem IBM-kompatiblen PC unter MS-DOS / Windows konzipiert. Im folgenden sind die Hard- und Softwarevoraussetzungen angegeben, wie sie bei der Entwicklung vorgesehen (und benutzt) wurden. Vermutlich läuft SALIERI auch auf 286-Rechnern mit ca.384K Arbeitsspeicher und unter älteren Betriebssystemversionen, dies wurde jedoch nicht getestet.

Empfohlene Hard- und Softwarekonfiguration:

- $\geq$ 386 Prozessor
- $\geq$ 640K Arbeitsspeicher (RAM)
- $\geq$ 1M freier Plattenspeicher
- Betriebssystem MS-DOS, Version $\geq$ 5.0

Um musikalische Objekte interaktiv wiedergeben zu können, werden außerdem benötigt:

- MIDI-fähige Soundkarte
- MS Windows, Version $\geq$ 3.1
- Automatischer Wiedergabeprozess („Wiedergabedämon“, siehe Anhang)

Da alle Ausgaben musikalischer Objekte im MIDI-Dateiformat erfolgen, sind weitere Programme, welche dieses Format verarbeiten können, oft nützlich. Insbesondere wird für ernsthaftere Anwendungen Software zur graphischen Ausgabe von MIDI-Dateien erforderlich sein.

## 2.2 Systemdateien

Das SALIERI-System umfaßt folgende Systemdateien:

| | |
|---|---|
| `SALIERI.EXE` | Systemprogramm |
| `SALIERI.WSP` | Workspace-Datei |
| `SALIERI.LOG` | Protokoll-Datei (zur Installation nicht benötigt) |

Darüberhinaus wird das System durch einige Programmbibliotheken und Demonstrationsprogramme ergänzt (`LIBS\*.SAL`, `DEMOS\*.SAL`), die jedoch zum Betrieb des Systems nicht unbedingt erforderlich sind.

Die Systemdateien sind in ein eigenes Verzeichnis auf der Festplatte zu kopieren (z.B. `C:\SALIERI`). Aus diesem Verzeichnis heraus sollte SALIERI jeweils aufgerufen werden.

Weiterhin sollte ein Verzeichnis für vom System verwaltete temporäre Dateien angelegt werden (z.B. `C:\SALIERI\TEMP`).

## 2.3 Umgebungsvariablen

Das SALIERI-System benutzt DOS-Umgebungsvariablen, um Verzeichnisse zum Lesen und Schreiben bestimmter Dateien festzulegen. Diese Umgebungsvariablen sollten vom Benutzer vor dem Aufruf von SALIERI gesetzt werden (z.B. in einer DOS-Kommandodatei). Folgende Umgebungsvariablen werden benutzt:

| | |
|---|---|
| `SALDIR` | Verzeichnis, in dem SALIERI-Dateien gesucht werden, wenn sie im aktuellen Verzeichnis nicht zu finden sind. |
| `SALTMPDIR` | Verzeichnis, in dem SALIERI temporäre Dateien erzeugt. |
| `SALPLYDIR` | Verzeichnis, in dem SALIERI direkt abzuspielende MIDI-Dateien erzeugt (dort kann der Wiedergabedämon auf sie warten). |

Diese Umgebungsvariablen enthalten jeweils einen einzelnen DOS-Pfad (z.B. `C:\SALIERI\TEMP`). Ist eine dieser Variablen nicht gesetzt, so wird für diese das aktuelle Verzeichnis angenommen.

## 2.4 Aufruf und Beendigung

Das SALIERI-System wird durch Aufruf der Datei `SALIERI.EXE` gestartet. Beim Aufruf können verschiedene Optionen angegeben werden:

`+p` persistenter Workspace (siehe Abschnitt 4.4), d.h. bei Systemstart wird der Systemzustand, der bei Beendigung der jeweils letzten Sitzung vorlag, wiederhergestellt. [Voreinstellung]

`-p` kein persistenter Workspace, d.h. bei jedem Systemstart wird SALIERI neu initialisiert, beim Beenden der Sitzung geht der Systemzustand verloren.

Weiterhin kann eine Liste von SALIERI-Dateien als Parameter angegeben werden, die dann nach dem Systemstart unmittelbar (in der angegebenen Reihenfolge) ausgeführt werden.

Um SALIERI zu verlassen, wird der Befehl `exit()` gegeben.

Im folgenden Beispiel wird das SALIERI-System (mit persistentem Workspace) aufgerufen und anschließend sofort wieder verlassen.

**Beispiel:**

```
C:\SALIERI> salieri +p
≡ ◇ ≡ Salieri - Version 2.0 ≡ ◇ ≡
(c)1995 by T. Helbich & H.H. Hoos

Loading workspace from salieri.wsp ... Done
> exit()
Saving workspace to salieri.wsp ... Done
See you later!

C:\SALIERI>
```

Die allgemeine Aufrufsyntax lautet:

```
salieri [+p | -p] [salfile] { salfile}
```

# 3 Grundsätzliches zur Bedienung

## 3.1 Kommandozeileneingabe

Die Benutzeroberfläche von SALIERI ist kommandozeilenorientiert (wie die von MS-DOS). Eingaben werden mit *Enter* abgeschlossen und dürfen das Zeilenende nicht überschreiten. Mit den Tasten $\uparrow, \downarrow$ können alte Eingaben abgerufen werden, zum Editieren der Kommandozeile dienen die Tasten $\leftarrow, \rightarrow$, *Home, End, Del* und *Backspace* (mit der üblichen Funktionalität). Durch Drücken von *Esc* wird die aktuelle Kommandozeile (unwiderruflich!) gelöscht.

Alle (mit *Enter* bestätigten) Eingaben einer Sitzung werden in der Datei `SALIERI.LOG` mitprotokolliert, diese Datei wird zum Beginn jeder Sitzung neu erzeugt, wobei der alte Inhalt verlorengeht.

Jede mit *Enter* bestätigte Eingabe wird vom SALIERI-Interpreter bearbeitet. Bei der Eingabe wird prinzipiell unterschieden zwischen Kommandos und Ausdrücken. Kommandos werden ausgeführt, wobei eventuell Ausgaben erfolgen (z.B. Bestätigungen oder Fehlermeldungen). Ausdrücke hingegen werden ausgewertet, wobei das Ergebnis in der Form *Result=...* ausgeben wird. Tritt während der Auswertung ein Fehler auf, wird eine Fehlermeldung (und kein Ergebnis) ausgegeben.

Die Funktion `it()` liefert jeweils das letzte von SALIERI ausgegebene Ergebnis einer Auswertung. Unmittelbar nach Systemstart ist `it()` undefiniert. :-)

**Beispiel:**

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

**Achtung:**

Parameterlose Funktionen und Kommandos (wie `exit` oder `ver`) müssen immer in der Form *name*`()` aufgerufen werden (siehe Abschnitt 7.2).

## 3.2 SALIERI-Dateien

Neben dem interaktiven Kommandozeilenbetrieb bietet das SALIERI-System die Möglichkeit, SALIERI-Programmdateien auszuführen. Dazu dient der Befehl `exec("`*filename*`")`, der die Programmdatei *filename*`.SAL` (aus dem aktuellen Arbeitsverzeichnis) ausführt.

SALIERI-Programmdateien enthalten eine Reihe von Kommandos, die jeweils durch „`;`“ voneinander getrennt sind (das Programm). Leerzeilen und zusätzliche Leerzeichen werden (wie üblich) ignoriert, Kommentare werden in der Form `(* ... *)` geschrieben und bei der Ausführung von Programmen vollständig ignoriert; sie können beliebig geschachtelt auftreten. Die Kommandos in einer SALIERI-Programmdatei werden gerade so ausgeführt, als wären sie von der Kommandozeile in der entsprechenden Reihenfolge eingegeben worden.

**Beispiel:** Programmdatei `DEMO1.SAL`

```
x := 10+2;              (* Objekt x wird erzeugt *)
stat();                 (* Systemzustand anzeigen *)
write();                (* Leerzeile ausgeben *)
write("Ergebnis: ",     (* Zeilenumbruch erlaubt!  *)
       2*(x-1));
```

**Beispiel:** Ausführung von `DEMO1.SAL`

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

x

229008 bytes of memory free

Ergebnis: 22
>
```

**Beachte:**

1. In Programmdateien dürfen Kommandos beliebig viele Zeilen (auch Leerzeilen) umfassen.
2. Anstelle des letzten Kommandos einer Programmdatei darf auch ein Ausdruck stehen, von der Verwendung dieser Möglichkeit wird jedoch abgeraten.
3. Das letzte Kommando einer Programmdatei muß nicht von einem Semikolon gefolgt sein, dennoch wird dies dringend empfohlen (um Fehlern bei späterer Erweiterung der Datei vorzubeugen).

Programmdateien können zur Realisierung von Bibliotheken verwandt werden, in diesem Falle besteht das Programm i.a. aus einer Reihe von Zuweisungen (siehe Abschnitt 4.1) und eventuell zusätzlichen Initialisierungskommandos.

Aus der Protokolldatei `SALIERI.LOG` kann durch Löschen aller Zeilen, die Ausdrücke enthalten, leicht eine korrekte Programmdatei erzeugt werden. Um diese ausführen zu können, muß sie allerdings zuvor in *filename*`.SAL` umbenannt werden.

# 4 Workspace und Objekte

Jedes Objekt besitzt einen Namen, einen Typ und einen Wert. Alle Objekte sind, nach ihren Namen in lexikographischer Reihenfolge geordnet im Workspace gespeichert (mit Ausnahme der internen Funktionen, siehe Abschnitt 7.5). Objektnamen beginnen immer mit einem Buchstaben oder Unterstrich („_“), gefolgt von beliebig vielen Buchstaben, Ziffern, Punkten („.“) oder Unterstrichen. Groß- und Kleinbuchstaben werden als verschieden angesehen.

Beispiel: Die Namen `x`, `x1`, `x.1`, `x_new.` und `x_new.1.a` sind gültige Objektnamen, während `1x` oder `.xyz` als Objektnamen nicht zulässig sind.

**Beachte:**

Objekte, deren Namen mit einem Unterstrich beginnen (wie `_x`), sind sogenannte Systemobjekte. Solche Objekte werden durch Ausgabekommandos wie `writeAll()` (siehe Abschnitt 4.2) im allgemeinen nicht angezeigt und sollten nur von erfahrenen Benutzern verwendet werden. Unerfahrene Benutzer sollten Systemobjekte weder erzeugen, noch ändern oder löschen.

## 4.1 Objekte erzeugen und löschen

Ein Objekt wird erzeugt, indem einem noch nicht vergebenen Namen ein Wert zugewiesen wird. Der Typ des Objekts ergibt sich dabei implizit aus dem Typen des zugewiesenen Wertes.

**Syntax der Wertzuweisung**

| *name* := *expression* |
|---|

Dabei ist *name* ein gültiger Name und *expression* ein beliebiger Ausdruck. Die einfachsten Ausdrücke sind Konstanten und Namen von Objekten. Weiterhin können Ausdrücke aus bereits bestehenden Ausdrücken zusammengesetzt werden, und zwar durch Anwendung von Funktionen bzw. Präfixoperatoren oder Verknüpfung mit Infixoperatoren. Schließlich können Ausdrücke geklammert werden.

**Syntax von Ausdrücken**

Ist $c$ eine Konstante, `x` ein beliebiger Objektname, `f` Name einer $n$-stelligen Funktion, „$\square$“ ein Präfixoperator, „$\diamond$“ ein Infixoperator und sind $exp_1, \ldots, exp_n$ gültige Ausdrücke, so sind genau die folgenden Ausdrücke ebenfalls gültig:

- $c$
- `x`
- `f(`$exp_1$, ... , $exp_n$`)`
- $\square$ $exp_1$
- $exp_1$ $\diamond$ $exp_2$
- `(`$exp_1$`)`

Die in SALIERI verfügbaren Konstanten, Funktionen und Operatoren werden in den folgenden Abschnitten eingeführt.

Wird einem bereits bestehenden Objekt ein Wert zugewiesen, so wird der alte Wert durch den neuen ersetzt, wobei der Typ automatisch angepaßt wird.

**Beispiel:** Zuweisung und Erzeugung von Objekten

```
>x.1 := 1
>x.1
Result= 1
>x.1 := "hello"
>x.1
Result= "hello"
>
```

Zum Löschen von Objekten dienen die Kommandos `delete(`*name*`)` und `deleteAll(`*maskStr*`)`, von denen das erste das Objekt mit Namen *name* aus dem Workspace löscht (falls vorhanden, sonst Fehlermeldung!). Der Befehl `deleteAll(`*maskStr*`)` löscht alle Objekte, auf deren Name die Schablone

*maskStr* paßt ("matching"). Diese Schablone ist eine Zeichenkette, die an beliebigen Stellen das Zeichen „`*`" enthalten darf. Dieses steht sozusagen stellvertretend für eine beliebige Zeichenkette (auch die leere, d.h. kein Zeichen!) an der entsprechenden Position des Objektnamens.

Beispiel: Die Schablone `"any*.*1"` paßt auf die Objektnamen `any.1`, `anyNewX.a1` und `anyNewX.main.a1`, nicht aber auf `any`, `Any.1` oder `anyNewX.a2`.

**Beispiel:** Löschen von Objekten

```
>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
>
```

*Bemerkung:* Beim Löschen von Objekten wird der entsprechende Speicherplatz im allgemeinen nicht sofort freigegeben (dieser Effekt wird durch die interne Objekt- und Speicherverwaltung bedingt).

## 4.2 Objekte inspizieren

Zum Auflisten der Namen aller im Workspace befindlichen Objekte dient das Kommando `stat()`. Dieses informiert außerdem über den noch verfügbaren Speicherplatz. Weiterhin können mithilfe des Kommandos `statAll(`*maskStr*`)` Informationen über alle Objekte, auf deren Namen die Schablone *maskStr* paßt, ausgegeben. Die Werte bestimmter Objekte können mithilfe der Befehle `write(`*name*`)` und `writeAll(`*maskStr*`)` ausgegeben werden.

Die Funktion `type(`*expr*`)` liefert den Typ des Ausdrucks *expr* als Zeichenkette. Insbesondere kann *expr* ein Objektname sein.

Zur Überprüfung, ob ein Objekt überhaupt im Workspace existiert, dient die Funktion `exists(`*name*`)`. Sie liefert `TRUE`, falls ein Objekt mit Namen *name* definiert ist, `FALSE` sonst.

**Beispiel:** Inspizieren von Objekten

```
>x.1:="hello"
>exists(x.1)
Result= TRUE
>type(x.1)
Result= "string"
>write(x.1)
hello
>
```

## 4.3 Speichern und Laden des Workspace

Eine Kopie des aktuellen Workspace kann mithilfe des Kommandos `save(`*filename*`)` in die Datei *filename*`.WSP` (im aktuellen Verzeichnis) gespeichert werden. Um den Inhalt einer solchen Workspace-Datei wieder in den aktuellen Workspace zu laden, wird das Kommando `load(`*filename*`)` aufgerufen. Dabei wird der aktuelle Workspace durch den in *filename*`.WSP` gespeicherten Workspace-Inhalt ergänzt. Falls ein Objektname sowohl im aktuellen Workspace als auch in der gespeicherten Workspace-Kopie benutzt wurde, so wird beim Laden der Workspace-Datei das entsprechende Objekt im aktuellen Workspace überschrieben.

## 4.4 Persistenz

Wenn beim Aufruf nicht anders spezifiziert (siehe Abschnitt 2.4), ist der Workspace persistent. Das bedeutet, daß bei Beendigung jeder Sitzung der Workspace gespeichert wird (in der Datei `SALIERI.WSP`); beim nächsten Start des SALIERI-Systems wird diese Workspace-Kopie wieder geladen, sodaß der alte Workspace-Inhalt zwischen zwei SALIERI-Sitzungen erhalten bleibt.

Dieser Mechanismus funktioniert allerdings nur, wenn die SALIERI-Sitzung ordnungsgemäß mit `exit()` beendet wurde. Um einen besseren Schutz gegen Verlust des Workspace-Inhalts durch Systemabsturz zu gewähren, ermöglicht das Kommando `sync()` eine Aktualisierung der Workspace-Datei `SALIERI.WSP` zu beliebigem Zeitpunkt. Dieser Befehl hat dieselbe Wirkung wie das Kommando `save("SALIERI.WSP")`.

# 5 Einfache Objekttypen

Obwohl das SALIERI-System vornehmlich zur Manipulation musikalischer Strukturen konzipiert wurde, benötigt es die üblichen einfachen Datentypen. Diese werden in den folgenden Abschnitten eingeführt und erläutert.

## 5.1 Wahrheitswerte

Wahrheitswerte werden durch den Objekttypen `"bool"` realisiert. Neben Konstanten `TRUE` und `FALSE` (für wahr und falsch) stehen die üblichen logischen Operationen `not, and, or` auf boolschen Ausdrücken (d.h. Ausdrücken mit Ergebnistyp `"bool"`) zur Verfügung:

**Boolsche Operationen**

| | |
|---|---|
| `not` *bool* | logische Negation |
| *bool* `and` *bool* | logisches und |
| *bool* `or` *bool* | logisches oder |

Für die boolschen Operationen gilt: `not` bindet stärker als `and`, dieses stärker als `or`.

*Beispiel:* Sind `x,y` Objekte vom Typ `"bool"`, so sind „`x and not y`“ sowie „`x and y or not x and not y`“ gültige boolsche Ausdrücke, von denen der zweite exakt dem vollgeklammerten Ausdruck „`(x and y) or ((not x) and (not y))`“ entspricht.

**Beispiel:** Boolsche Operationen

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

## 5.2 Integer

Der Typ `"int"` realisiert ganze Zahlen zwischen $-32768$ und 32767. Die Konstanten werden als gewöhnliche Zahlen mit Vorzeichen geschrieben („+“ kann weggelassen werden, also $+256 = 256$). Auf Objekten dieses Typs sind die üblichen arithmetischen Operationen definiert (Addition, Subtraktion, Multiplikation, gerundete Division, Divisionsrest), zusätzlich Funktionen zur Betragsbildung und Erzeugung gleichverteilter Zufallszahlen in einem vorgegebenen Intervall.

**Integeroperationen**

| | |
|---|---|
| `+`*int* | positives Vorzeichen |
| `-`*int* | negatives Vorzeichen |
| *int* `+` *int* | Addition |
| *int* `-` *int* | Subtraktion |
| *int* `*` *int* | Multiplikation |
| *int* `div` *int* | Ganzzahldivision (`7 div 3 = 2`) |
| *int* `mod` *int* | Divisionsrest (`7 mod 3 = 1`) |
| `abs(`*int* `)` | Absolutbetrag |
| `rndUniform(`$int_1$`,`$int_2$`)` | liefert Zufallszahl $i$ mit $int_1 \leq i \leq int_2$ |

Es gilt: Vorzeichen binden stärker als `*,div,mod`, diese binden stärker als `+,-`. Ansonsten werden die Ausdrücke von links nach rechts ausgewertet, d.h. „`-1 + 11 div 5 mod 3`“ entspricht „`(-1) + ((11 div 5) mod 3)`“ und ergibt bei Auswertung `1`.

**Achtung:**

Überschreitungen des Zahlbereichs (z.B. bei Addition oder Multiplikation) verursachen keine Fehlermeldungen, führen aber in der Regel zu völlig falschen Ergebnissen.

**Beispiel:** Integeroperationen

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

## 5.3 Reals

Gleitpunktzahlen werden in SALIERI durch den Typen `"real"` realisiert. Die Konstanten werden mit Dezimalpunkt und Vorzeichen geschrieben („+“ kann weggelassen werden, also +12.3 = 12.3). Auf Objekten dieses Typs sind neben den üblichen arithmetischen Operationen (Addition, Subtraktion, Multiplikation, Division), Funktionen zur Betragsbildung und zur Erzeugung gleichverteilter Zufallszahlen definiert. Weiterhin können Gleitpunktzahlen durch Auf- oder Abrunden in Ganzzahlen konvertiert werden.

Die umgekehrte Umwandlung von Ganz- in Gleitpunktzahlen kann durch Multiplikation mit `1.0` bewirkt werden. Weiterhin können überall, wo Gleitpunktzahlen erwartet werden, auch Integer verwendet werden; diese werden dann automatisch konvertiert (z.B. im Audruck `5/2`, der zu `2.5` auswertet).

Vom SALIERI-System werden Gleitpunktzahlen immer mit 8 Nachkommastellen ausgegeben (wobei die letzte Stelle gerundet wird), intern wird jedoch im allgemeinen mit weiteren Stellen gerechnet.

**Realoperationen**

| | |
|---|---|
| `+`*real* | positives Vorzeichen |
| `-`*real* | negatives Vorzeichen |
| *real* `+` *real* | Addition |
| *real* `-` *real* | Subtraktion |
| *real* `*` *real* | Multiplikation |
| *real* `/` *real* | Division |
| `abs(`*real*`)` | Absolutbetrag |
| `rnd()` | liefert Zufallszahl $r$ mit $0 < r < 1$ |
| `floor(`*real* `)` $\rightsquigarrow$ *int* | Abrunden (untere Gaußklammer) |
| `ceil(`*real* `)` $\rightsquigarrow$ *int* | Aufrunden (obere Gaußklammer) |
| `root(`$real_1$`,` $real_2$ `)` $\rightsquigarrow$ *real* | Wurzel von $real_2$ zur Basis $real_1$ |
| `root(`*real*`)` $\rightsquigarrow$ *real* | Quadratwurze |
| l `exp(`$real_1$`,` $real_2$ `)` $\rightsquigarrow$ *real* | $real_1$ potenziert mit $real_2$ |
| `exp(`*real*`)` $\rightsquigarrow$ *real* | die Eulersche Zahl potenziert mit *real* |
| `log(`$real_1$`,` $real_2$ `)` $\rightsquigarrow$ *real* | Logarithmus von $real_2$ zur Basis $real_1$ |
| `log(`*real*`)` $\rightsquigarrow$ *real* | natürlicher Logarithmus |
| `sin(`*real*`)` $\rightsquigarrow$ *real* | Sinus (im Bogenmaß) |
| `cos(`*real*`)` $\rightsquigarrow$ *real* | Cosinus (im Bogenmaß) |
| `tan(`*real*`)` $\rightsquigarrow$ *real* | Tangens (im Bogenmaß) |
| `arctan(`*real*`)` $\rightsquigarrow$ *real* | Arcustangens (im Bogenmaß) |

Es gilt: Vorzeichen binden stärker als `*,/`, diese binden stärker als `+,-`. Ansonsten werden die Ausdrücke von links nach rechts ausgewertet.

**Beispiel:** Realoperationen

```
>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
>
```

**Achtung:**

- Die Gleitpunktarithmetik ist nicht immer genau. Besonders bei betragsmäßig großen Zahlen können Ungenauigkeiten in den hinteren Stellen auftreten.
- Überschreitungen des Zahlbereichs führen in der Regel zum Absturz des SALIERI-Systems (dabei geht der gesamte Inhalt des Workspace verloren!).

## 5.4 Zeichenketten

Zur Realisierung von Zeichenketten dient in SALIERI der Typ `"string"`. Die Konstanten dieses Typs werden als `"..."` geschrieben, wobei zwischen den Anführungszeichen eine beliebige Zeichenkette stehen darf, die kein Anführungszeichen enthält. Auf Zeichenketten ist die Operation der Konkatenation („Aneinanderhängen“) definiert. Weiterhin stehen Funktionen zum Extrahieren, Einfügen und Löschen von Teilketten zur Verfügung.

Außerdem gibt es Funktionen zur Bestimmung der Länge einer Zeichenkette (in Zeichen) sowie zum Auffinden bestimmter Teilketten. Die Positionen innerhalb einer Zeichenkette der Länge $n$ sind mit 1 bis $n$ numeriert, sodaß in `"J.S.Bach"` (Länge 8) an Position 3 das Zeichen `"S"` steht. Zeichenketten können prinzipiell eine Länge von bis zu 32767 Zeichen erreichen.

**Stringoperationen**

| | |
|---|---|
| *string* `+` *string* | Konkatenation („Aneinanderhängen“) |
| `infix(`*string*`,` $int_1$`,`$int_2$`)` | liefert Ausschnitt von *string*, der bei $int_1$ beginnt und $int_2$ Zeichen lang ist; für positive $int_2$ wird nach rechts gezählt, für negative nach links. $int_2$ kann ausgelassen werden und es wird dann standardmäßig ein Ausschnitt der Länge 1 geliefert. |
| `ins(`$string_1$`,` $string_2$`,`*int*`)` | liefert Zeichenkette, die durch Einfügen von $string_2$ in $string_1$ unmittelbar vor Position *int* entsteht; *int* muß zwischen 1 und `length(`$string_1$`)+1` liegen. |
| `del(`*string*`,` $int_1$`,`$int_2$`)` | liefert Zeichenkette, die aus *string* durch Löschen von $int_2$ Zeichen, beginnend mit dem Zeichen an Position $int_1$ entsteht, für positive $int_2$ wird nach rechts gezählt, für negative nach links. $int_2$ kann ausgelassen werden und es wird dann standardmäßig nur ein Zeichen gelöscht. |
| `length(`*string*`)` $\rightsquigarrow$ *int* | liefert Länge von *string* in Zeichen. |
| `posNext(`$string_1$`,`$string_2$`,`*int*`)` $\rightsquigarrow$ *int* | liefert Position des nächsten Auftretens von $string_2$ in $string_1$, wobei ab Position *int* nach rechts gesucht wird; falls kein Auftreten gefunden wurde, wird der Wert 0 zurückgegeben. *int* kann ausgelassen werden und es wird dann standardmäßig ab der ersten Position gesucht. |

*Schreibweise:* Für `length(`*string*`)` kann gleichbedeutend `#`*string* geschrieben werden. Der Operator „`#`“ bindet dabei stärker als „`*`“ bzw. „`/`“. Auch `infix(`*string*`,` $int_1$`,`$int_2$`)` kann abgekürzt werden: *string*($int_1$,$int_2$). $int_2$ ist wieder optional. Diese Form von `infix` ist jedoch nur auf deklarierte String-Objekte anwendbar, Konstanten führen zu einer Fehlermeldung.

**Beachte:**

- Die Funktionen `infix` und `del` akzeptieren auch Längenangaben $int_2$, die über die letzte (bzw. erste) Position der Zeichenkette hinausragen, in diesem Falle wird die Längenangabe so interpretiert, als würde sie gerade bis zum Beginn (bzw. Ende) der Zeichenkette reichen.
- Wird beim Aufruf von `ins` oder `del` eine ungültige Position `int` bzw. $\texttt{int}_1$ angegeben, so liefern diese Funktionen die unveränderte Zeichenkette `string` zurück.
- Werden `infix` bzw. `posNext` mit ungültigen Positionen $int_1$ bzw. *int* aufgerufen, so wird als Ergebnis `""` bzw. `0` zurückgegeben.

Diese Operationen sind ebenfalls für andere Objekttypen definiert, die reihenförmig organisiert sind (Notensequenzen und Listen) und werden daher Reihenoperationen genannt. Wegen der vielfältigen Verwendbarkeit der Reihenoperationen werden diese im folgenden Beispiel ausführlich verdeutlicht.

**Beispiel:** Reihenoperationen auf Zeichenketten

```
>name := "A.Salieri";
>piece := "Klavierkonzert B-Dur";
>name + ", " + piece
Result= "A.Salieri, Klavierkonzert B-Dur"
>#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, 16, 5)
Result= "Klavierkonzert "
>del(name, 2, -2)
Result= "Salieri"
>ins(piece, "in ", 16)
Result= "Klavierkonzert in B-Dur"
>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")
>
```

## 5.5 Vergleichsoperationen

In SALIERI stehen sechs Vergleichsoperatoren zur Verfügung: „`=`“ und „`!=`“ (Gleichheit und Ungleichheit) sind für alle einfachen und musikalischen Objekttypen (siehe Abschnitt 6) definiert; für Zahlen und Zeichenketten existieren weiterhin die Operatoren „`<`“, „`>`“, „`<=`“ und „`>=`“ (diese beziehen sich bei Zeichenketten auf die lexikographische Ordnung).

Alle Vergleichsoperatoren sind Infixoperatoren mit Ergebnistyp `"bool"` und binden stärker als die logischen Operatoren, aber schwächer als alle anderen Operatoren (demnach sind die Ausdrücke

„`2*x+5 > 10 and not x > 5`“ und „`(((2*x)+5) > 10) and (not (x > 5))`“ gleichbedeutend).

*Beachte:* Vergleiche können nur zwischen Objekten gleichen Typs ausgeführt werden, insbesondere sind Vergleiche wie „`1/4 = "hello"`“ ungültig (und liefern nicht etwa `FALSE`!). Vergleiche zwischen ganzen Zahlen und Gleitpunktzahlen sind wegen der automatischen Umwandlung von *int* nach *real* erlaubt.

**Beispiel:** Vergleichsoperationen

```
>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 "!=")
>
```

# 6 Musikalische Objekttypen

Das SALIERI-System dient in erster Linie zur Erzeugung und Manipulation musikalischen Materials. Musikalische Strukturen werden prinzipiell auf Notenebene dargestellt, hierzu stellt SALIERI zwei Objekttypen zur Verfügung: Notensequenzen (`"sequence"`) zur Realisierung einstimmiger Musik und Notensegmente (`"segment"`) zur Darstellung mehrstimmigen Materials.

In den folgenden Abschnitten werden zunächst die Konstanten für Notensequenzen und -segmente eingeführt, sodann einige grundlegende Operationen auf musikalischen Objekten erläutert und anschließend die verfügbaren musikalischen Operationen vorgestellt.

## 6.1 Sequenzkonstanten

Ein musikalischer Ton wird im SALIERI-System durch 5 Parameter beschrieben: die sogenannten Hauptparameter Tonhöhe und relative Tondauer sowie die Nebenparameter Intensität, Klangfarbe und Metrum.

Eine vollständige Tonbeschreibung hat die Form „`M=`$m$ `T=`$t$ `I=`$i$ $qr$`*`$e$`/`$d$ “. Dabei bedeuten die Symbole $m, t, i, q, r, e, d$ im einzelnen:

| | |
|---|---|
| $m$ | Metrum in 1/4-Noten pro Minute (Metronomzahl), ganze Zahl zwischen 1 und 255 |
| $t$ | Klangfarbe in Form einer Programmnummer eines MIDI-Instrumentes, ganze Zahl zwischen 0 und 127 |
| $i$ | (relative) Intensität, ganze Zahl zwischen 0 und 127 (logarithmische Teilung!) |
| $q$ | Tonqualität, entweder einer der Notennamen `c,d,e,f,g,a,h` gegebenenfalls direkt gefolgt von einem Vorzeichen `#` oder `&`; oder „`_`“ für Pausen, zusätzlich können auch die chromatischen Bezeichner `cis,dis,fis,gis,ais` oder die italienischen Notennamen `do,re,mi,fa,sol,la,si` verwendet werden (siehe dazu auch „Systemeinstellungen ändern“, `set("noteNames",...)`) |
| $r$ | Register (Oktave), ganze Zahl zwischen -4 und +6 |
| $e$ | Zähler der relativen Tondauer, ganze Zahl zwischen 1 und 128 |
| $d$ | Nenner der relativen Tondauer, ganze Zahl zwischen 1 und 128 |

Die Tondauer, repräsentiert durch $e/d$, kann zusätzlich durch Hinzufügen von Punkten beeinflußt werden. Es können punktierte und doppelt punktierte Notenwerte angegeben werden. Die Punktierung muß sich direkt hinter der Dauerangabe befinden.

*Beispiel:* „`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`“ und „`M=80 T=10 I=40 c*1/8.`“ sind vollständige Tonbeschreibungen.

Notensequenzkonstanten werden als Folgen von Tonbeschreibungen in der Form `[`$n_1$ $n_2$ `...]` notiert. Dabei muß in den Tonbeschreibungen $n_1, n_2$ usw. immer der Tonqualitätsparameter $q$ angegeben werden, alle anderen Angaben können weggelassen werden. Fehlt eine solche Angabe, so wird sie von der vorhergehenden Note übernommen (Ausnahme: die relative Tondauer wird nur übernommen, wenn weder $e$ noch $d$ angegeben werden. Bei unvollständigen Angaben wie `/8` oder `*2` wird $e = 1$ bzw. $d = 1$ gesetzt. Punktierungen müssen immer explizit angegeben werden.). Fehlen Parameterangaben für die erste Note einer Sequenz, so werden folgende Standardeinstellung benutzt:

- für die Oktavlage $r = 1$
- für die Tondauer $e/d = 1/4$
- für das Metrum $m = 100$
- für die Klangfarbe $t = 0$ (entspricht normalerweise Klavier)
- für die Intensität $i = 80$ (entspricht normalerweise mezzoforte)

Die Standardwerte für Metrum, Klangfarbe und Intensität können den eigenen Vorlieben angepasst werden. Siehe dazu „Systemeinstellungen ändern“.

Wenn das System Notensequenzen ausgibt, werden die Hauptparameter für jede Note vollständig angegeben, die Nebenparameter jedoch nur am Anfang einer Sequenz und dort, wo ihr Wert sich ändert. Das Ausgabeformat von Sequenzen kann über mehrere Systemparameter beeinflußt werden, hierzu dient das Systemkommando `set()` (siehe „Systemeinstellungen ändern“).

**Beispiel:** Ausgabeformat von Notensequenzen

```
>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 ]
>
```

Die Vergleichsoperatoren „`=`“ und „`!=`“ sind auf Notensequenzen anwendbar, dabei sind zwei Notensequenzen genau dann gleich, wenn sie die gleiche Länge haben und alle auftretenden Töne bezüglich ihrer Hauptparameter (Tonhöhe und -dauer) gleich sind.

Notensequenzen können prinzipiell eine Länge von bis zu 32767 Tonbeschreibungen erreichen. Notensequenzen mit einer Länge von mehr als 1000 Tönen sind allerdings (wegen der langen Wartezeiten bei der Verarbeitung und des hohen Speicherbedarfs) kaum von praktischem Nutzen.

**Beachte:**

- Pausen werden prinzipiell nicht zusammengefaßt.
- Aus technischen Gründen werden lange Sequenzen nicht als zusammenhängende Sequenzkonstante ausgegeben, sondern in der Form `[...] + [...] + ··· + [...]`.
- `c#`, `d&` und `cis` werden als unterschiedliche Tonqualitäten betrachtet. Zum Beispiel liefert so der Vergleich `[c#]=[cis]` ein *FALSE* als Ergebnis.

## 6.2 Segmentkonstanten

Ein Segment besteht aus einer Reihe von Notensequenzen, den Stimmen. Alle Stimmen eines Segments beginnen zum gleichen Zeitpunkt und laufen dann parallel ab. Die Konstanten des Typs `"segment"` können auf zwei verschiedene Arten geschrieben werden.

In einfach polyphoner Form sind sie von der Form $\{seq_1, seq_2, \ldots\}$ , wobei $seq_1$ usw. Notensequenzen (in der oben angegebenen Form) sind.

*Beispiel:* `{[c d e], [e f g]}` und `{[c d e]}` sind Segmente in einfach polyphoner Form.

Weiterhin können Segmente als Akkordsequenzen in der Form $[a_1\ a_2\ \ldots]$ geschrieben werden, dabei sind $a_1$ usw. entweder Tonbeschreibungen (s.o.) oder Akkordbeschreibungen der Form $\{n_1, n_2, \ldots\}$ (mit Tonbeschreibungen $n_1$ usw.).

*Beachte:* Nicht spezifizierte Tonparameter werden auch innerhalb eines Akkordes von der letzten Note übernommen, sodaß `[{c2/8, e, g}]` als `[{c2*1/8, e2*1/8, g2*1/8}]` interpretiert wird.

*Beispiel:* `[{c,e,g} {h0,d1,g}]` und `[{c,e,g}]` sind Segmente in Akkordsequenzform.

Notensegmente werden vom SALIERI-System immer in einfach polyphone Normalform transformiert, in dieser Form besteht ein Segment aus einer Reihe paralleler Notensequenzen, die alle die gleiche Dauer haben. Um diese Form zu erreichen, werden in der einfach polyphonen Form an die Stimmen entsprechende Pausen angehängt. Akkordsequenzen werden zunächst in Einzelstimmen aufgelöst, indem die jeweils ersten Töne jedes Akkords die erste Stimme bilden usw. (dabei wird durch Einfügen von Pausen gewährleistet, daß die Noten eines jeden Akkords auch tatsächlich gemeinsam beginnen.

**Beispiel:** Transformation in einfach polyphone Normalform

```
>{[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 ])
>
```

**Beachte:**

- Die Kombination von einfach polyphoner und Akkordsequenzform in Segmentkonstanten (etwa `{[{c,e,g} {h0,d1,g}]}` oder `[{[c e g]}]`) ist nicht zulässig.
- Die Reihenfolge der Stimmen in einem Segment ist nicht bedeutungslos, es gilt: $\{seq_1, seq_2\} \neq \{seq_2, seq_1\}$
  (Gleiches gilt erst Recht für die Reihenfolge der Töne in einem Akkord, da diese insbesondere über die Zuordnung zu den Stimmen entscheidet).

Die Vergleichsoperatoren „`=`“ und „`!=`“ sind auf Notensegmente anwendbar, dabei sind zwei Segmente genau dann gleich, wenn sie aus gleichen Stimmen in der gleichen Reihenfolge bestehen (siehe Gleichheit von Notensequenzen).

Die (hypothetische) Obergrenze für die Anzahl von Stimmen innerhalb eines Segments liegt bei 32767. Für die MIDI-Ausgabe können allerdings nur 16 vollständig unabhängige Stimmen genutzt werden (werden mehr Stimmen verwendet, müssen MIDI-Kanäle mehrfach belegt werden, wodurch z.B. die Klangfarbenzuordung gestört werden kann).

## 6.3 Basisoperationen auf Sequenzen

Auf Notensequenzen sind als Basisoperationen die Reihenfunktionen `+`, `infix`, `ins`, `del`, `length` und `posNext` definiert (vergl. Abschnitt 5.4; beachte: `posNext` vergleicht beim Suchen nur die Hauptparameter!). Diese wirken auf Sequenzen ebenso wie auf Zeichenketten, wobei die „Basisobjekte“ hier Tonbeschreibungen statt Zeichen sind. Weiterhin ist eine Funktion `retro(`*seq*`)` verfügbar, die zu einer gegebenen Notensequenz die zeitliche Umkehrung liefert („Krebs“). Diese Funktion kann auch auf Segmente angewendet werden, in diesem Falle wird jede Stimme einzeln umgekehrt.

**Beispiel:** Reihenoperationen auf Notensequenzen

```
>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 ]
>
```

Als grundlegende Variationsoperationen werden Transposition und (zeitliche) Expansion durch die Funktionen `transp` und `expand` realisiert:

**Variation der Hauptparameter:**

| | |
|---|---|
| `transp`(*seq*, *int*) | liefert die um *int* Halbtonschritte nach oben transponierte Sequenz *seq*. |
| `expand`(*seq*, $int_1$, $int_2$) | liefert eine zeitlich um den Faktor $int_1/int_2$ „gestreckte“ (bzw. „gestauchte“) Variante von *seq*; genauer wird hierbei jede relative Tondauer der Gestalt $e/d$ durch $(e * int_1)/(d * int_2)$ ersetzt. |

Eine Transposition nach unten ist durch Angabe einer negativen Anzahl von Halbtonschritten zu erreichen. Die Funktionen `transp` bzw. `expand` verändern jeweils nur Tonhöhen bzw. relative Tondauern der gegebenen Sequenz, alle anderen Tonparameter bleiben erhalten.

**Beispiel:** Transposition und 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 ]
>
```

Die Basisoperationen zur Variation der Nebenparameter (Tonintensität, Metrum und Klangfarbe) sind `cresc`, `accel` und `instr`. Diese sind folgendermaßen definiert:

**Variation der Nebenparameter:**

| | |
|---|---|
| `cresc(`*seq*`,` $int_1$`,` $int_2$`)` | liefert die Sequenz, die aus *seq* ensteht, wenn diese mit einem linearen Intensitätsverlauf von $int_1$ nach $int_2$ versehen wird. |
| `accel(`*seq*`,` $int_1$`,` $int_2$`)` | liefert die Sequenz, die aus *seq* ensteht, wenn diese mit einem linearen Metrumsverlauf von $int_1$ nach $int_2$ versehen wird. |
| `instr(`*seq*`,` *int*`)` | liefert die Sequenz, die aus *seq* ensteht, wenn deren Klangfarbe durchgängig auf *int* gesetzt wird. |

Bei diesen Funktionen ist darauf zu achten, daß zulässige Werte für die auftretenden Intensitäts-, Metrums- und Klangfarbenwerte angegeben werden (s.o.). Durch Anwendung der Funktionen `cresc` bzw. `accel` wird ein eventuell vorhandener Intensitäts- bzw. Metrumsverlauf vollständig überschrieben.

**Beispiel:** Operationen zur Variation der Nebenparameter

```
>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 ]
>
```

Die Funktionen `transp`, `expand`, `cresc`, `accel`, und `instr` können auch auf Segmente angewendet werden, in diesem Falle wird die enstprechende Operation für jede einzelne Stimme durchgeführt.

## 6.4 Basisoperationen auf Segmenten

Wie bereits erwähnt, besteht ein Notensegment aus einer Reihe von Sequenzen, den Stimmen. Die Funktion `parts(`*segm*`)` liefert zu einem Segment *segm* die Anzahl der Stimmen, aus denen dieses besteht. Für `parts(`*segm*`)` kann gleichbedeutend `#`*segm* geschrieben werden. Die Stimmen eines Segments sind mit $1, \ldots, n$ numeriert; die Funktion `part(`*segm*`,`*int*`)` liefert Stimme *int* von *segm* (oder die leere Sequenz `[]`, falls $int > n$ oder $int < 1$).

**Beispiel:** Basisoperationen auf Notensegmenten (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 ]
>
```

Die beiden wichtigsten Operationen auf Notensegmenten sind `+` und `poly`. Die Funktion `poly(`$s_1$, ... ,$s_n$`)` kombiniert $s_1$, ... ,$s_n$ zu einem neuen Segment, in welchem sämtliche Einzelstimmen der $s_1$,

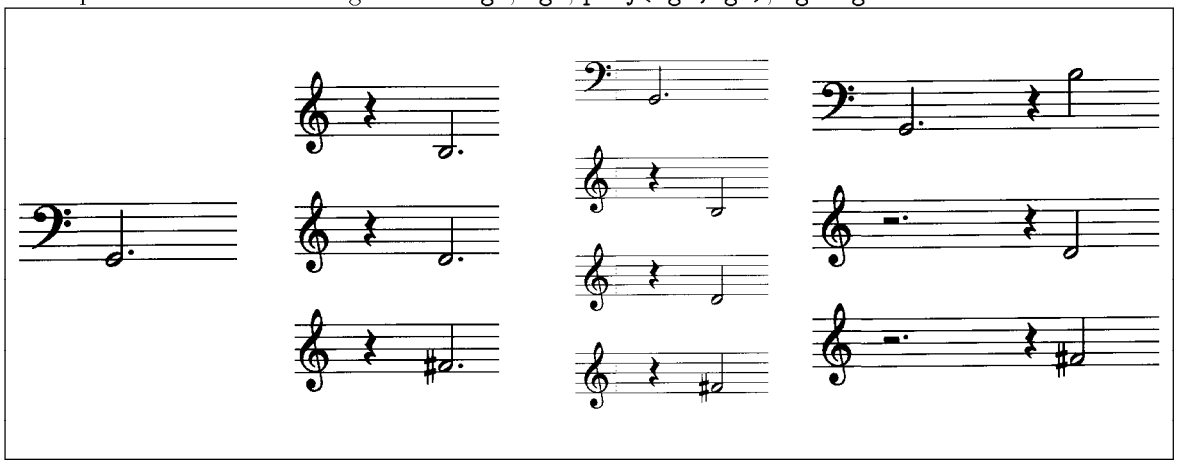
$\ldots, s_n$ (gleichzeitig beginnend) parallel ablaufen. Durch Auffüllen der einzelnen Stimmen mit Pausen am Ende wird gewährleistet, daß alle Stimmen des neuen Segments gleiche Dauer haben. Die $s_1, \ldots, s_n$ stehen dabei für beliebige Segmente oder Sequenzen (in diesem speziellen Fall werden Sequenzen wie entsprechende einstimmige Segmente behandelt!). Abkürzend kann für `poly(`$s_1, \ldots, s_n$`)` auch { $s_1, \ldots, s_n$} benutzt werden.

Mittels des Operators `+` können Segmente (zeitlich) „aneinandergehängt“ werden. Dies geschieht für $s_1$`+`$s_2$, indem die erste Stimme von $s_2$ an die erste Stimme von $s_1$ angehängt die erste Stimme des neuen Segmentes ergibt usw. (Beachte: Alle Stimmen eines Segmentes haben prinzipiell gleiche Dauer!). Ist die Stimmenanzahl von $s_1$ und $s_2$ unterschiedlich, so wird diese zunächst durch „Auffüllen mit leeren Stimmen“ angeglichen.

**Beispiel:** Basisoperationen auf Notensegmenten (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 ])
>
```

Basisoperationen auf Notensegmenten: `sg1`, `sg2`, `poly(sg1,sg2)`, `sg1+sg2`



## 6.5 Musikalische Deskriptorfunktionen

Deskriptorfunktionen liefern Informationen über musikalische Objekte. In SALIERI stehen folgende Funktionen dieser Art zur Verfügung:

**Deskriptorfunktionen:**

| | |
|---|---|
| `isRest(`*seq*`,` *int*`)` $\rightsquigarrow$ *bool* | liefert `TRUE` genau dann, wenn die Note an Position *int* von *seq* eine Pause ist, `FALSE` sonst. Der Parameter *int* kann ausgelassen werden, es wird dann standardmäßig Position 1 angenommen. |
| `getPitch(`*seq*`,` *int*`)` $\rightsquigarrow$ `LIST(`$int_1$`,` $int_2$`,` $int_3$`)` | liefert die Tonhöhe des Tons an Position *int* von *seq* als Liste (vergl. Abschnitt 9.1) aus Tonqualität ($int_1$; dabei stehen $0, 1, \ldots$ für $c, c\#, \ldots$) und Oktave ($int_2$); $int_3$ gibt die Erhöhung bzw. Erniedrigung durch Vorzeichen an. Die Tonqualität $int_1$ wurde durch Erniedrigung einer Note um Halbtonschritte erreicht, wenn $int_3$ negativ ist, bzw. durch Erhöhung, wenn $int_3$ positiv ist. Der Betrag von $int_3$ bestimmt die Anzahl der Halbtonschritte der Alteration. Beginnt *seq* mit einer Pause, wird als Tonqualität $-1$ zurückgegeben. Der Parameter *int* kann ausgelassen werden, es wird dann standardmäßig Position 1 angenommen. |
| `getPitchTonal(`$seq_1$`,` $seq_2$`,` *int*`)` $\rightsquigarrow$ `LIST(`$int_1$`,` $int_2$`)` | liefert die relative Tonhöhe einer Note an Position *int* der Sequenz $seq_1$ bezüglich einer frei definierbaren Tonleitersequenz $seq_2$. Der Parameter *int* kann ausgelassen werden, es wird dann standardmäßig Position 1 angenommen. $int_1$ beschreibt die Stufe der gefragten Note, die sie in Bezug auf die Tonleiter $seq_2$ einnimmt. Die erste Note der Tonleiter hat den Stufenwert 0, die zweite den Stufenwert 1 und so weiter. Tiefere Noten, als die in der Tonleitersequenz definierten, erhalten entsprechend negative Werte. $int_2$ gibt die Erhöhung bzw. Erniedrigung durch Vorzeichen an. |
| `ambitus(`*seq*`)` $\rightsquigarrow$ *int* | liefert den Tonumfang (also den Abstand zwischen höchstem und tiefstem Ton) der Sequenz *seq* in Halbtonschritten. |
| `duration(`*seq*`)` $\rightsquigarrow$ `LIST(`$int_1$`,` $int_2$`)` | liefert die relative Dauer $int_1/int_2$ der Sequenz *seq* als Liste aus Zähler und Nenner; die relative Dauer einer Sequenz ergibt sich als der Summe der relativen Dauern der einzelnen Töne und Pausen. |
| `time(`*segment*`)` $\rightsquigarrow$ *real* | liefert die absolute Dauer des Segments *segment* in Sekunden; im Gegensatz zur relativen Dauer wird hier der Metrumsparameter berücksichtigt. Auch auf Sequenzen anwendbar. |
| `getIntens(`*seq*`,` *int*`)` $\rightsquigarrow$ *int* | liefert die Intensität der Note an Position *int* der Sequenz *seq*. |
| `getTempo(`*seq*`,` *int*`)` $\rightsquigarrow$ *int* | liefert den Wert des Metrums für die Note an Position *int* der Sequenz *seq*. |
| `getInstr(`*seq*`,` *int*`)` $\rightsquigarrow$ *int* | liefert den Wert des Klangfarbenparameters für die Note an Position *int* der Sequenz *seq*. |

Mithilfe der Funktion `getPitch` können leicht Intervalle bestimmt werden, ähnliches gilt für Tondauerverhältnisse („rhythmische Intervalle“) unter Verwendung der Funktion `duration`.

**Beispiel:** Musikalische Deskriptorfunktionen

```
>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
>
```

## 6.6 Musikalische Variationsfunktionen

Salieri stellt zwei Funktionen zur Variation der Tonhöhenfolge von Segmenten oder Sequenzen zur Verfügung: `invert` und `intvExpand`; diese bewirken eine Umkehrung bzw. Streckung (oder Stauchung) der Intervallstruktur. Diese Funktionen sind folgendermaßen definiert:

**Intervallfolgenvariation:**

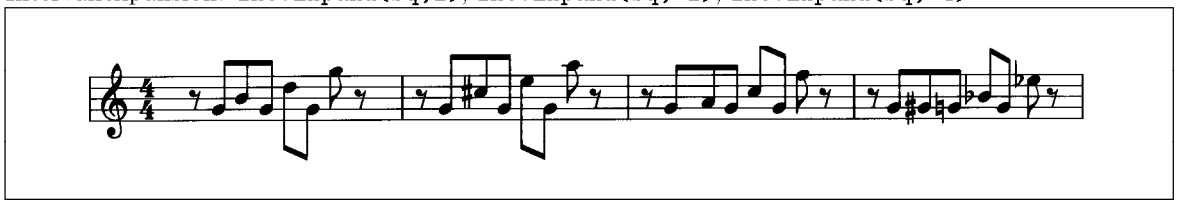
| | |
|---|---|
| `invert(`*seq*`)` | liefert die Sequenz, die durch Umkehren aller Intervalle aus *seq* entsteht („Spiegel“); der Anfangston und die Intervallgrößen bleiben dabei erhalten, die Intervallrichtungen werden umgekehrt. |
| `intvExpand(`*seq*`,` *int*`)` | liefert die Sequenz, die aus *seq* ensteht, wenn alle Intervalle um *int* Halbtonschritte vergrößert werden; die Intervallrichtungen bleiben dabei erhalten, kleine Sekunden können nicht mehr weiter verkleinert werden. |

Pausen werden bei diesen Intervalloperationen prinzipiell ignoriert; wird bei `intvExpand` eine negativen Anzahl von Habltonschritten angegeben, so findet eine Verkleinerung der Intervalle statt.

**Beispiel:** Intervallfolgenvariation

```
>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 ]
>
```

Intervallexpansion: `intvExpand(sq,2)`, `intvExpand(sq,-2)`, `intvExpand(sq,-4)`



Neben der Funktion `transp` stehen zwei weitere Transpositionsfunktionen zur Verfügung: `transpTonal` und `transcr`, diese bewirken tonale Transposition bzw. Transkription. Beide Funktionen benötigen Tonleitern als Parameter. Eine Tonleiter ist eine Notensequenz, in der keine Tonqualität mehrfach vorkommt (und die keine Pausen enthält). Gewöhnlicherweise sind außerdem die Töne nach aufsteigender Tonhöhe geordnet und der Tonumfang beträgt weniger als eine Oktave.

*Beispiel:* `[c d e f g a h]` ist eine Tonleiter in diesem Sinne (C-Dur), ebenso `[c g d2 a2]` (keine konventionelle Tonleiter); `[c d e f g a h c2]` ist hingegen keine Tonleiter, da die Tonqualität *c* doppelt auftritt.

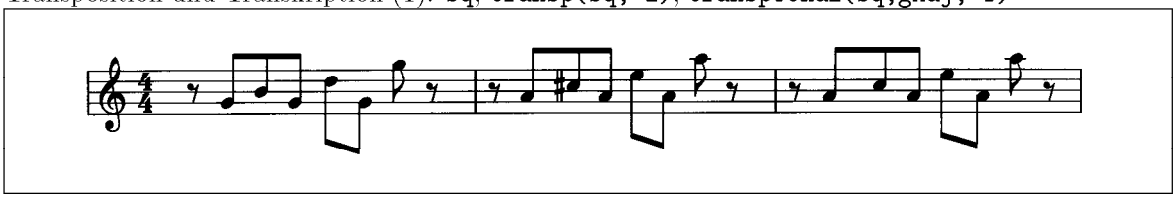
Die tonale Transposition besteht in einer „Verschiebung“ der Tonhöhen entlang der angegebenen Tonleiter um eine gewisse Anzahl von Stufen. Die angegebene Tonleiter wird dabei ggf. in beide Richtungen durch geeignete Oktavierung fortgesetzt. Problematisch ist die Behandlung von „leiterfremden“ Tönen, also von Tonqualitäten, die in der angegeben Tonleiter nicht auftreten; in der gegenwärtigen Realisierung werden diese zu Pausen entsprechender Dauer umgewandelt. Die chromatische Transposition (`transp`) kann als Spezialfall der tonalen Transposition aufgefaßt werden, dabei wird als Tonleiter die chromatische Tonleiter `[c c# d d# ...a# h]` verwendet.

Bei Anwendung der Transkriptionsoperation werden sozusagen die Töne einer gegebenen Sequenz von einer Tonleiter in eine andere „übersetzt“. Dabei wird für jeden Ton zunächst dessen Position in der Ausgangstonleiter bestimmt; der Ton wird dann durch den Ton an der entsprechenden Position der Zieltonleiter ersetzt. Die angegebenen Tonleiter werden wiederum ggf. in beide Richtungen durch geeignete Oktavierung fortgesetzt; leiterfremde Töne bleiben unverändert. Die tonale Transposition ist ein Spezialfall der Transkription, bei der die Zieltonleiter durch die entsprechend „verschobene“ Ausgangstonleiter gegeben ist.

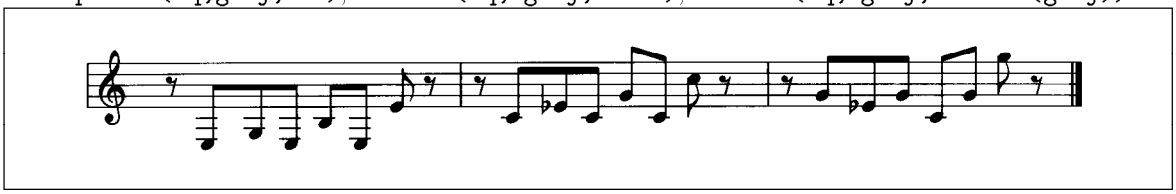
**Erweiterte Transpositionsfunktionen:**

| | |
|---|---|
| `transpTonal(`*seq*$_1$`,` *seq*$_2$`,` *int*`)` | liefert die Sequenz, die durch tonale Transposition um *int* Leiterstufen nach oben aus *seq*$_1$ ensteht, dabei wird als Tonleiter *seq*$_2$ zugrundegelegt (s.o.). |
| `transcr(`*seq*$_1$`,` *seq*$_2$`,` *seq*$_3$`)` | liefert die Sequenz, die durch Transkription der Sequenz *seq*$_1$ von Tonleiter *seq*$_2$ in Tonleiter *seq*$_3$ ensteht (s.o.). |

Transposition und Transkription (1): `sq`, `transp(sq,+2)`, `transpTonal(sq,gMaj,+1)`



Transposition und Transkription (2):
`transpTonal(sq,gMaj, -9)`, `transcr(sq, gMaj, cMin)`, `transcr(sq, gMaj, invert(gMaj))`



**Beispiel:** Erweiterte Transpositionsfunktionen

```
>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 ]
>
```

Zur Änderung der Alteration gegebener Sequenzen können die folgenden Funktionen dienen.

**Alterationsfunktionen:**

| | |
|---|---|
| `alter(`*seq*`,` *int*`)` | liefert die um *int* alteriert dargestellte Sequenz *seq*, wobei ein positiver Wert *int* entsprechend viele Kreuzvorzeichen, ein negativer b-Vorzeichen erzeugt. Ist *int* gleich 0, wird die Sequenz ohne Vorzeichen dargestellt. Gegebenenfalls wird dabei auf chromatische Notennamen zurückgegriffen. |
| `alterTonal(`*seq*`,` *int*`)` | liefert ebenso wie `alter` eine alteriert dargestellte Sequenz, jedoch ohne Benutzung chromatischer Notenbezeichner. Es wird jede Note aus *seq* mit diatonischen Notennamen und soweit möglich mit einer *int* entsprechenden Anzahl Vorzeichen alteriert dargestellt. |

Die Töne der Sequenz behalten also ihre klingende Tonhöhe, werden aber jeweils mit einer gewünschten Zahl von Vorzeichen notiert.

**Beachte:**

- Die Einstellung des Systemparameters `noteNames` beeinflußt die Darstellung der Notenkonstanten innerhalb einer Sequenz. Um die mit `alter` gewählte Anzahl Vorzeichen auch bei schon in der Ausgangssequenz alterierten und/oder als Zwischentöne (`cis`, `dis`, ...) deklarierten Noten dargestellt zu bekommen, muß `noteNames` auf `chromatic` eingestellt sein (zur Einstellung der Systemparameter siehe „Systemeinstellungen ändern“).
- `alterTonal` transformiert zusätzlich zur Alteration die intern als Zwischentöne repräsentierten Noten in ihre entsprechende diatonische Repräsentation. Dies kann z.B. dazu benutzt werden, die bei Tonleitertranspositionen entstehenden unschönen Zwischentonrepräsentationen zu beseitigen.

**Beispiel:** Alterationsfunktionen

```
>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 ]
>
```

Die Funktion `fragm(`*seq*`,`*real*$_1$`,`*real*$_2$`)` extrahiert einen bestimmten zeitlichen Abschnitt eines Stückes. Sie liefert das Fragment der Sequenz *seq*, das an relativer Zeitposition *real*$_1$ beginnt und eine relative Dauer von *real*$_2$ hat. Ist der Wert für *real*$_1$ kleiner 0 oder größer als die relative Dauer von *seq*, so wird die leere Sequenz zurückgegeben. Bei negativer relativer Dauer *real*$_2$ gibt die Zeitposition *real*$_1$ den rechten Rand (also das Ende) des Fragments an, die Dauer beträgt dann *real*$_2$. Überschreitet das spezifizierte Fragment den rechten (oder linken) Rand von *seq*, so wird es entsprechend gekürzt (und nicht etwa mit Pausen aufgefüllt!).

**Beispiel:** Extraktion zeitlicher Fragmente

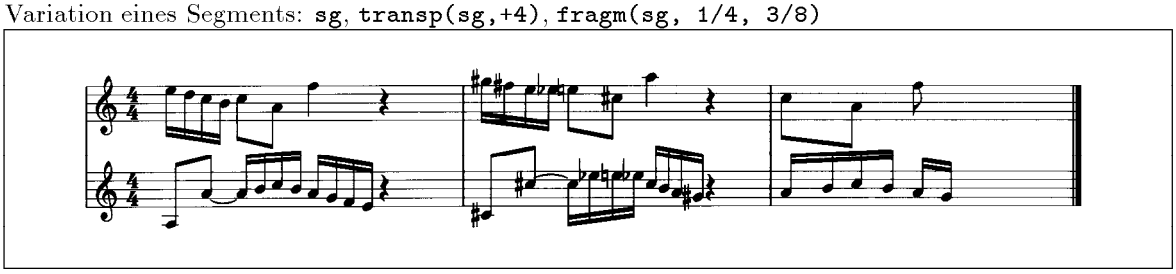
```
>sq := [_/8 g h g d2 g1 g2]
>fragm(sq, 5/16, 2/8)
Result= [ I=80 M=100 T=0 h1/16 g1/8 d2/16 ]
>fragm(sq, 2/4, 8/4)
Result= [ I=80 M=100 T=0 d2/8 g1/8 g2/8 ]
>fragm(sq, 2/4, -8/4)
Result= [ I=80 M=100 T=0 _/8 g1/8 h1/8 g1/8 ]
>
```

Alle genannten Variationsfunktionen können auch auf Segmente angewendet werden, in diesem Falle wird die enstprechende Operation für jede einzelne Stimme durchgeführt.

**Beispiel:** Variationsfunktionen auf Segmenten

```
>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 eines Segments: `sg`, `transp(sg,+4)`, `fragm(sg, 1/4, 3/8)`



## 6.7 Musikalische Kombinationsfunktionen

SALIERI stellt zwei Funktionen zur Kombination von Sequenzen zur Verfügung: `comb` und `merge`. Diese sind folgendermaßen definiert:

**Kombinationsfunktionen:**

| | |
|---|---|
| `comb(`$seq_1$`,` $seq_2$`)` | liefert die Sequenz, die durch Kombination der Tonhöhenfolge von $seq_1$ mit der Tondauerfolge von $seq_2$ ensteht; die jeweils kürzere Sequenz wird dabei ggf. wiederholt; die Nebenparameter werden auf die Standardeinstellungen gesetzt. |
| `merge(`$seq_1$`,` $seq_2$`)` | liefert die Sequenz, die durch Anhängen der Sequenz $seq_2$ an $seq_1$, wobei der letzte Ton $t_1$ von $seq_1$ mit dem ersten Ton $t_2$ von $seq_2$ verschmolzen wird; dabei wird die Dauer von $t_1$ um die Dauer von $t_2$ verlängert und $t_2$ gelöscht. |

Die Funktion `comb` eignet sich gut, um eine gegebene Melodie mit einem sich wiederholenden rhythmischen Motiv zu unterlegen. Die `merge`-Operation bildet sozusagen ein Gegenstück zu `fragm`; bei Anwendung von `merge` werden die relativen Dauern der Sequenzen summiert, aber jeweils eine Tonhöhe (nämlich die von $t_2$, s.o.) geht verloren.

**Beispiel:** Kombinationsfunktionen

```
>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 ]
>
```

# 7 Benutzerdefinierte Funktionen

Das SALIERI-System bietet dem Benutzer die Möglichkeit, neue Funktionen zu definieren und so das System seinen Bedürfnissen gemäß zu erweitern. Funktionen werden dabei als Objekte behandelt und damit prinzipiell genauso wie die Objekte anderer Typen (Zahlen, Sequenzen etc.) verwaltet. Prinzipiell wird je nach der Behandlung der Parameter zwischen Funktionen und Makros unterschieden. Am Ende dieses Abschnitts werden auch die internen Funktionen kurz besprochen, welche ebenfalls über einen eigenen Objekttyp realisiert sind.

## 7.1 Funktionsdefinition

Funktionen werden durch eine Anweisung der Form *name* `:= FUNC(`*int*`,` *body*`)` definiert; dabei ist *int* die Anzahl der Parameter, der Funktionskörper *body* eine durch `'...'` geklammerte Anweisungsfolge und der Funktionsname *name* ein gültiger Objektname.

*Bemerkung:* Die Funktionsdefinition ist eine echte Wertzuweisung (vergl. Abschnitt 4.1), wobei `FUNC` ein Konstruktor für Funktionsobjekte ist.

Innerhalb des Funktionskörpers werden die Parameter gemäß ihrer Position in der Parameterliste beim Funktionsaufruf als `$1`, `$2` usw. angesprochen. Jede Funktion hat einen eigenen (lokalen) Workspace. Dieser enthält zu Beginn der Funktionsausführung alle Objekte des globalen Workspace (als Kopien!) und die Funktionsparameter und wird am Ende der Funktionsausführung automatisch gelöscht. Wird also ein Objekt innerhalb einer Funktion erzeugt, so existiert dieses nur bis zum Ende des entsprechenden Funktionsaufrufs; Objekte des globalen Workspace können innerhalb von Funktionen zwar beliebig verwendet, aber nicht dauerhaft geändert oder gelöscht werden. Die Benennung lokaler Objekte ist daher unkritisch.

Zum Beenden einer Funktion und zur Rückgabe eines Ergebnisses dient der Befehl `return`. Durch die Anweisung `return(`*expr*`)` wird die Ausführung der Funktion an dieser Stelle beendet (d.h. eventuell folgende Anweisungen werden nicht mehr ausgeführt!), wobei der Wert des Ausdrucks *expr* als Funktionsergebnis zurückgegeben wird. Die Anweisung `return()` wirkt genauso, außer daß in diesem Falle die Funktion kein Ergebnis liefert (sie ist in diesem Sinne also eher ein Kommando).

**Beispiel:** Funktionsdefinition

```
scaleMaj := FUNC(1,
  (* WIRKUNG: liefert die Durtonleiter auf einem gegebenen Grundton
     AUFRUF: scaleMaj(<note>)
     BSP: 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));
  ');
```

**Beachte:**

- Funktionsdefinitionen sollten prinzipiell in Programmdateien geschrieben werden; auf diese Weise sind Änderungen leicht durchführbar.
- Funktionen sind gewöhnliche Objekte, d.h. sie können insbesondere in Zuweisungen oder als Parameter von Funktionen auftreten.
- Die syntaktische Korrektheit von Funktionskörpern wird prinzipiell erst bei Ausführung der entsprechenden Funktion geprüft (nicht etwa bei der Definition, siehe auch Abschnitt 12).
- Funktionen können rekursiv definiert sein, d.h. insbesondere sich selbst in ihrem Funktionskörper aufrufen. Die Rekursionstiefe ist durch den verfügbaren Speicherplatz beschränkt. ACHTUNG: Zu tiefe oder endlose Rekursionen können zum Systemabsturz führen (Datenverlust!).
- Für Funktionsparameter wird keine automatische Typprüfung vorgenommen. Damit ist es möglich, Funktionen zu realisieren, die auf Objekten verschiedener Typen operieren (Polymorphismus). Der Anwender hat dafür Sorge zu tragen, daß bei der Ausführung einer benutzerdefinierten Funktion keine Typkonflikte entstehen. Es empfiehlt sich dringend, für häufiger benutzte Funktionen die zulässigen Parametertypen als Kommentar zu vermerken bzw. eine explizite Typprüfung durchzuführen (vergl. Beispiel).
- Obwohl Funktionsparameter wie `$1` gewöhnliche Objekte des lokalen Workspace sind, sollten sie nicht überschrieben oder gar gelöscht werden. Ebenfalls hilfreich ist es, bei längeren Funktionen die Parameter durch entsprechende Zuweisungen am Anfang des Funktionskörpers sinnvoll zu benennen.
- Als Ergebnis können prinzipiell beliebige Objekte zurückgegeben werden. Vorsicht ist jedoch geboten, falls eine Funktion `f1` ein Funktionsobjekt `f2` als Ergebnis liefert. In diesem Fall dürfen im Körper von `f2` keine lokalen Objekte von `f1` auftreten, da diese nur definiert sind, während `f1` ausgeführt wird.
- Funktionsdefinitionen können geschachtelt werden, d.h. lokale Funktionen können innerhalb eines Funktionskörpers definiert werden. Aus Gründen der Übersichtlichkeit sollte dies jedoch im allgemeinen vermieden werden.

*Bemerkung:* Die Vergleichsoperatoren „`=`“ und „`!=`“ sind auf Funktionen anwendbar, dabei sind zwei

Funktionen genau dann gleich, wenn sie die gleiche Parameteranzahl sowie zeichenweise identische Funktionskörper haben.

## 7.2 Funktionsaufruf

Benutzerdefinierte Funktionen werden wie interne Funktionen in der Form *name*`(`$par_1$`, ..., `$par_n$`)` aufgerufen. Dabei ist *name* der Name der Funktion und $par_1$ usw. sind die Funktionsparameter. Als Parameter können beliebige Ausdrücke auftreten. Es gibt auch Funktionen, die keine Parameter besitzen, in diesem Falle hat der Aufruf die Form *name*`()`.

Liefert die entsprechende Funktion ein Resultat zurück, so ist der Funktionsaufruf ein Ausdruck, der stellvertretend für dieses Resultat steht. Wird kein Funktionswert zurückgegeben, handelt es sich bei dem Funktionsaufruf um ein Kommando; dieser darf folglich nicht auf der rechten Seite einer Zuweisung oder als Funktionsparameter verwendet werden.

**Beispiel:** Funktionsaufruf

```
>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 ]
>
```

*Bemerkung:* Eine alternative Form des Funktionsaufrufs benutzt die interne Funktion `apply`, dabei entspricht der Aufruf `apply(`*name*`,` $par_1$`, ...,` $par_n$`)` genau der oben angegebenen Aufrufform *name*`(`$par_1$`, ...,` $par_n$`)`. Im allgemeinen sollte die Verwendung von `apply` aus Gründen der Übersichtlichkeit vermieden werden.

## 7.3 Makros

Makros sind spezielle Funktionen, die durch eine Anweisung der Form *name* `:= MACRO(`*int*`,` *body*`)` definiert werden. Der einzige Unterschied zu gewöhnlichen Funktionen besteht darin, daß Variablen in Makros nicht zwangsläufig lokal sind. Lokale Variablen müssen in Makros mit einem $-Zeichen beginnen. Diese Notation ist auch in Funktionsdeklarationen erlaubt, bringt dort aber keine zusätzliche Funktionalität. Alle Objektnamen ohne führendes $, die in einem Makrokörper auftauchen, beziehen sich also auf den globalen Workspace und sind mit entsprechender Vorsicht zu behandeln (ungewollte Seiteneffekte durch Überschreiben von Objekten).

Makros eignen sich besonders zur Realisierung eigener Systemkommandos, wie beispielsweise dem Anzeigen oder Löschen bestimmter Objektgruppen.

**Beispiel:** Makrodefinition

```
setMyScales := MACRO(1,
  (* WIRKUNG: setzt die Variablen myT,myD,myS auf die Durtonleitern
              der Tonika, Dominante und Subdominante zum gegebenen Grundton
     AUFRUF: setMyScales(<note>)
     BSP: setMyScales([c]) *)
  `myT := scaleMaj($1);
  myD := transp(myT,+7);
  myS := transp(myT,+5);
  ');
```

**Beispiel:** Makroaufruf

```
>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
>
```

*Bemerkung:* Die Vergleichsoperatoren „=“ und „!=“ sind auf Makros anwendbar, dabei sind zwei Makros genau dann gleich, wenn sie die gleiche Parameteranzahl sowie zeichenweise identische Funktionskörper haben. Makros und Funktionen sind vergleichbar und werden prinzipiell als verschieden (ungleich) angesehen.

## 7.4 Funktionen mit variabler Parameteranzahl

Oft ist es bequem, wenn eine benutzerdefinierte Funktion mit unterschiedlich vielen Parametern aufgerufen werden kann. In SALIERI kann dies realisiert werden, indem bei der Funktionsdefinition als Parameteranzahl $-1$ angegeben wird. In diesem Falle werden beim Aufruf alle Parameter zu einer Liste zusammengefaßt, auf die innerhalb des Funktionskörpers mit dem Namen `$1` zugegriffen wird. Eine solche Funktion kann mit beliebig vielen Parametern aufgerufen werden (auch ohne Parameter, dann ist die Parameterliste `$1` leer).

**Beispiel:** Definition

```
demoVarPar := FUNC(-1,
  (* AUFRUF: demoVarPar(<any_1>, ..., <any_n>)
     BSP: demoVarPar(1,2,3, [c d e], [c# d# f]) *)
  `parlist := $1;
  write("Anzahl Parameter: ", #parlist);
  write("Parameterliste: ", parlist);
  ');
```

**Beispiel:** Aufruf

```
>demoVarPar(1,2,3, [c d e], [c# d# f]);
Anzahl Parameter: 5
Parameterliste: 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 ])
>
```

*Bemerkung:* Makros mit variabler Parameteranzahl können selbstverständlich auf die gleiche Weise realisiert werden.

## 7.5 Interne Funktionen

Interne Funktionen werden ebenso als Objekte betrachtet, wie benutzerdefinierte Funktionen oder Makros. Insbesondere können sie also ebenfalls in Zuweisungen oder als Funktionsparameter auftreten.

Diese Eigenschaft kann beispielsweise dazu genutzt werden, „Kopien“ von Systemkommandos unter einem neuen Namen anzulegen, und so das System eigenen Bedürfnissen anzupassen.

**Beispiel:** Interne Funktionen

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

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

**Beachte:**

Interne Funktionen können nicht umdefiniert oder gelöscht werden. Beim Zugriff auf ein beliebiges Objekt wird zunächst nach einem internen Funktionsobjekt des entprechenden Namens gesucht, erst wenn ein solches nicht vorhanden ist, wird im Workspace nachgeschaut.
So ist es zwar möglich, beispielsweise ein Objekt namens `exit` zu definieren, beim Aufruf von `exit` wird aber trotzdem auf die (vordefinierte) interne Funktion zurückgegriffen.

*Bemerkung:* Die Vergleichsoperatoren „`=`“ und „`!=`“ sind auf interne Funktionen anwendbar, zwei interne Funktionsobjekte sind genau dann gleich, wenn sie dieselbe interne Funktion bezeichnen. Interne Funktionsobjekte sind mit Makros und Funktionen vergleichbar und werden prinzipiell als verschieden (ungleich) von diesen angesehen.

# 8 Kontrollstrukturen

Übliche Kontrollstrukturen wie Verzweigung oder Schleifen werden hauptsächlich zur Ablaufkontrolle innerhalb benutzerdefinierter Funktionen benötigt und sind in SALIERI als Funktionen realisiert.

In diesem Abschnitt werden zunächst die entsprechenden vordefinierten Funktionen `if`, `while`, `loop` und `loopn` vorgestellt. Abschließend wird kurz darauf eingegangen, wie im Rahmen des Systems neue Kontrollstrukturen realisiert werden können.

## 8.1 Blöcke

Blöcke sind Folgen von SALIERI-Anweisungen, welche durch `'...'` geklammert sind. Genaugenommen handelt es sich dabei um parameterlose Makros, die Schreibweisen `'...'` und `MACRO(0,'...')` sind daher gleichbedeutend.

Es wird zwischen Anweisungsblöcken und Ausdrucksblöcken unterschieden: erstere enthalten lediglich Anweisungen, letztere enthalten anstelle der letzten Anweisung einen Ausdruck (oder eine `return`-Anweisung) und ergeben bei Auswertung somit einen Ergebniswert.

Beispiel: `'x := x+1'` und `'x := y; y := f(x); write(y)'` sind Anweisungsblöcke; `'2 > 1'` und `'x := x+1; return(x > 5)'` sind Ausdrucksblöcke (von denen der erstere bei Auswertung `TRUE` ergibt). `''` ist ebenfalls ein gültiger Anweisungsblock, er ist allerdings bei Ausführung völlig wirkungslos. `'x < 1; x > 5'` ist kein gültiger Block, da er zwei durch „;“ verbundene Ausdrücke enthält (was prinzipiell nicht zulässig ist).

*Bemerkung:* Blöcke werden beispielsweise zur Realisierung von Schleifen als Kontrollfunktionen benötigt, da in SALIERI Funktionsparameter prinzipiell strikt, d.h. vor dem Funktionsaufruf, ausgewertet werden (die Auswertung einer Anweisungsfolge entspricht ihrer Ausführung). Die Zusammenfassung von Anweisungen zu einem Block verhindert die unmittelbare Ausführung dieser Anweisungsfolge. Die Ausführung eines Blockes wird in der Regel durch die Funktion `apply` bewirkt (vergl. Abschnitt 7.2).

## 8.2 Verzweigung

Die Kontrollfunktion `if` realisiert die bedingte Ausführung von Anweisungen. Sie wird im allgemeinen als Anweisung in einer der Formen `if(`*cond-block*`,` *if-block*`)` oder `if(`*cond-block*`,` *if-block*`,` *else-block*`)` verwendet.

Dabei ist *cond-block* ein Block, der bei Auswertung einen Wahrheitswert liefert; *if-block* ist ein Anweisungsblock der nur dann ausgeführt wird, wenn die Auswertung von *cond-block* den Wert `TRUE` ergibt. Der Anweisungsblock *else-block* muß nicht angegeben werden, ist er vorhanden, so wird er nur dann ausgeführt, wenn die Auswertung von *cond-block* nicht den Wert `TRUE` ergibt.

**Beispiel:** Verzweigung

```
>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 ]
>
```

*Bemerkung:* Die Parameter *if-block* und *else-block* der `if`-Funktion können auch Ausdrucksblöcke sein (allerdings nur beide gemeinsam!), in diesem Falle wird der gesamte Funktionsaufruf zu einem Ausdruck. Diese Möglichkeit sollte jedoch nur von erfahrenen Anwendern genutzt werden.

Mit `if` können auch mehrere bedingte Anweisungen ohne umständliche Verschachtelungen strukturiert werden (ähnlich dem von C bekannten *case*-Konstrukt). Es wird hierzu die Form `if(`*cond-block*$_1$`,` *if-block*$_1$`,` *cond-block*$_2$`,` *if-block*$_2$`, ... ,` *else-block*`)` verwendet. *else-block* ist auch hier optional.

**Beispiel:** Mehrstufige Verzweigung

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

## 8.3 Schleifen

SALIERI stellt drei Schleifenanweisungen zur Verfügung: `while`, `loop` und `loopn`. Diese Anweisungen werden in der folgenden Form verwendet:

**Syntax der Schleifenanweisungen:**

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

Dabei ist *cond-block* jeweils ein Audrucksblock mit Ergebnistyp `"bool"` und *body-block* ein Anweisungblock, der den Schleifenkörper enthält.

Bei Ausführung einer `while`-Schleife wird, solange die Auswertung von *cond-block* den Wert `TRUE` ergibt, der Schleifenkörper *body-block* ausgeführt.

**Beispiel:** `while`-Schleife

```
>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 ]
>
```

Die `loop`-Schleife ist die allgemeinste Schleifenkonstruktion (sie entspricht der for-Schleife in der Programmiersprache C). Wird eine `loop`-Anweisung ausgeführt, so wird zunächst (einmalig) der Anweisungsblock *init-block* ausgeführt, der im allgemeinen Anweisungen zur Initialisierung der Schleife enthält (etwa Initialisierung von Zählvariablen). Anschließend wird, solange die Auswertung von *cond-block* den Wert `TRUE` ergibt, zunächst der Schleifenkörper *body-block* und direkt anschließend der Anweisungsblock *iter-block* ausgeführt; letzterer sollte Anweisungen zur Kontrolle der Schleife (etwa Änderung von Zählvariablen) enthalten.

**Beispiel:** `loop`-Schleife

```
>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 ]
>
```

Die `loopn`-Anweisung bewirkt ein *int*-maliges Ausführen des Schleifenkörpers *body-block* falls *int* eine ganze Zahl (oder entsprechender Ausdruck) größer Null ist (ansonsten ist die Anweisung wirkungslos).

**Beispiel:** `loopn`-Schleife

```
>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 ]
>
```

**Beachte:**

- Aus Gründen der Übersichtlichkeit sollte *cond-block* prinzipiell nur einen boolschen Ausdruck enthalten (und keine Nebenwirkungen verursachen)
- Die Funktionalität der `while`- und `loopn`-Schleife läßt sich auch mithilfe der `loop`-Anweisung erreichen. Allerdings wird `loopn` im allgemeinen wesentlich schneller ausgeführt als eine gleichbedeutende Formulierung mit `while`; gleiches gilt für `while` und `loop`. Dennoch sollte bei der Wahl des jeweils zu verwendenden Schleifenkonstrukts (soweit möglich) immer der übersichtlichsten und verständlichsten Lösung der Vorzug gegeben werden.

## 8.4 Benutzerdefinierte Kontrollstrukturen

Da in SALIERI die Übergabe von Funktionen (und damit auch von Blöcken) als Funktionsparameter erlaubt ist, können weitere Kontrollstrukturen auf einfache Weise als benutzerdefinierte Funktionen realisiert werden. Die als Parameter übergebenen Blöcke werden dabei mithilfe der Funktion `apply` ausgewertet (im Falle von Ausdrucksblöcken) bzw. ausgeführt (im Falle von Anweisungsblöcken).

Solche Kontrollfunktionen sollten, wie alle benutzerdefinierten Funktionen sorgfältig dokumentiert werden (Kommentare!).

Das folgende Beispiel realisiert eine `until`-Schleife, deren Körper sooft ausgeführt wird, bis die Abbruchbedingung erfüllt ist (mindestens jedoch einmal).

**Beispiel:** Definition der `until`-Schleife

```
doUntil := MACRO(2,
  (* WIRKUNG: führt anweisungsblock $1=<block> sooft aus, bis ausdrucks-
              block $2=<cond> TRUE ergibt
     AUFRUF: doUntil(<block>, <cond>)
     BSP: doUntil('i := i+1', 'i > 10') *)
  'apply($1);
  while ('not (apply($2) = TRUE)','apply($1)');
  ');
```

**Beispiel:** Verwendung der `until`-Schleife

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

Im abschließenden Beispiel wird ein konditionaler Ausdruck realisiert, der abhängig von einem boolschen Wert verschiedene Ergebnisse liefert.

**Beispiel:** Definition des konditionalen Ausdrucks

```
cond := MACRO(3,
  (* WIRKUNG: liefert Wert $2=<if-expr>, falls die Auswertung des boolschen
              Ausdrucks $1=<cond-expr> TRUE ergibt, Wert $3=<else-expr> sonst
     AUFRUF: cond(<cond-expr>, <if-expr>, <else-expr>)
     BSP: cond(n > 0, n, -n) *)
  `if(`$1=TRUE',
     `return($2)',
     `return($3)'
     );
  ');
```

**Beispiel:** Verwendung des konditionalen Ausdruck

```
>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 ]
>
```

# 9 Datenstrukturen

Neben den einfachen und musikalischen Datentypen bietet SALIERI die Möglichkeit, beliebige Objekte zu (heterogenen) Listen zusammenzufassen. Eine solche Liste kann gleichzeitig Objekte verschiedener Typen enthalten; auf diese Weise lassen sich viele Datenstrukturen realisieren (Beachte: eine Liste kann als Element auch eine weitere Liste enthalten, auf diese Weise sind Baumstrukturen repräsentierbar).

In diesem Abschnitt wird zunächst der Objekttyp `"list"` mit den zugehörigen grundlegenden Operationen eingeführt. Anschließend wird gezeigt, wie unter Verwendung von Listenobjekten andere Datenstrukturen realisiert werden können.

## 9.1 Listen

Listen werden durch Ausdrücke der Form `LIST(`$obj_1$`, ..., `$obj_n$`)` erzeugt; dabei sind $obj_1$ usw. beliebige Objekte (oder Ausdrücke, die Objekte beschreiben). Abkürzend kann der Konstruktor `LIST` weggelassen werden. Der einelementige Fall (z.B `(3)`) ist die Ausnahme, da die als geklammerter Ausdruck und nicht als einelementige Liste interpretiert wird. Speziell läßt sich die leere Liste als `()` beschreiben.

*Beispiel:*
`LIST(1,2,3)`, (1,2,3), `LIST([c/4 e g],0.25,"test")` und `LIST( LIST(1/4,TRUE), FUNC(1,`'`return($1*2)`'`)` sind gültige Listen. Insbesondere ist `LIST()` oder `()` eine gültige Liste (die leere Liste).

Für Listen sind neben den Reihenoperationen `+`, `infix`, `ins`, `del`, `length` und `posNext` (siehe Ab-

schnitt 5.4) noch fünf weitere Basisoperationen definiert:

**Listenfunktionen:**

| | |
|---|---|
| `LISTINIT(`*int*`,` *obj*`)` | liefert die Liste, die *int*-mal das Objekt *obj* enthält. |
| `consList(`*obj*`,` *list*`)` | liefert die Liste, die durch Einfügen des Objekts *obj* in die Liste *list* vor Position 1 ensteht. |
| `getList(`*list*`,` *int*`)` | liefert Element an Position *int* der Liste *list* (die Zählung beginnt bei 1). |
| `putList(`*list*`,` *obj*`,` *int*`)` | liefert die Liste, die aus *list* entsteht, wenn das Element an Position *int* durch das Objekt *obj* ersetzt wurde. |
| `inList(`*list*`,` *obj*`)` $\rightsquigarrow$ *bool* | liefert `TRUE` genau dann, wenn die Liste *list* das Objekt *obj* als Element enthält, `FALSE` sonst. |

*Schreibweisen:* Für `length(`*list*`)` kann gleichbedeutend `#`*list* geschrieben werden; der Operator „`#`“ bindet dabei stärker als „`*`“ bzw. „`/`“. Für `getList(`*list*`,` *int*`)` kann gleichbedeutend *list*`@`*int* geschrieben werden; der Operator „`@`“ bindet dabei stärker als jeder andere Operator.

**Beispiel:** Listenoperationen

```
>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 ]
>
```

Die Vergleichsoperatoren „`=`“ und „`!=`“ sind auf Listen anwendbar, dabei sind zwei Listen genau dann gleich, wenn sie die gleiche Länge haben und alle Elemente gleich sind.

Listen können prinzipiell eine Länge von bis zu 32767 Elementen erreichen.

## 9.2 Realisierung weiterer Datenstrukturen

Auf der Basis des Listenkonzepts lassen sich praktisch beliebige weitere Datenstrukturen realisieren. Die Abbildung auf die zugrundeliegende Listenstruktur erfolgt dabei durch einen Satz von Zugriffsfunktionen. Da die Listen, welche die neue Datenstruktur realisieren im allgemeinen eine bestimmte Struktur aufweisen müssen, sollte innerhalb der Zugriffsfunktionen sichergestellt werden, daß diese

Struktur erhalten bleibt.

Im folgenden Beispiel wird die Implementation einer dynamischen record-Struktur demonstriert. Ein record ist ein Datensatz, der aus einer Menge von Feldern, auf die durch Angabe von Feldnamen zugegriffen wird. Jeder Datensatz wird als Liste realisiert, die jeweils abwechselnd einen Feldnamen und den zugehörigen Feldinhalt enthält. Auf dieser Datenstruktur sind folgende Operationen möglich: Erzeugen eines leeren Datensatzes, Überschreiben eines Feldes, Auslesen eines Feldes, Zuweisung ganzer Datensätze (Kopieren) sowie Löschen eines Datensatzes. Löschen und Kopieren können in der üblichen Weise durchgeführt werden, die übrigen Operationen werden als Funktionen realisiert und in einer Bibliothek zusammengefaßt.

**Beispiel:** Bibliothek `RECORD.SAL`

```
(* SALIERI 2.0
   BIBLIOTHEK record.sal
   INHALT: Implementation des abstrakten Datentypen "record"
   FUNKTIONEN: newRec, putRec, getRec *)

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

newRec := FUNC(-1,
  (* WIRKUNG: erzeugt record mit feldnamen <str_1>, ..., <str_n>
     AUFRUF:  newRec(<str_1>, ..., <str_n>) -> record
     BSP:     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,
  (* WIRKUNG: liefert <record>, nachdem Inhalt des Feldes <str>
              auf <any> gesetzt wurde
     AUFRUF:  putRec(<record>,<str>,<any>) -> <record>
     BSP:     putRec(myrec,"title","Präludium 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,
  (* WIRKUNG: liefert Inhalt des Feldes <str> von <record>,
     AUFRUF:  getRec(<record>,<str>) -> <any>
```

```
   BSP:     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("")'
  );
');
```

Diese Bibliothek kann folgendermaßen benutzt werden:

**Beispiel:** Verwendung der Bibliothek `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","Präludium in C","composer","J.S.Bach","date","")
>write(getRec(myRec,"title"));
Präludium in C
>write(getRec(myRec,"composer"));
J.S.Bach
>
```

# 10 Ein- und Ausgabefunktionen

Um SALIERI-Programmen die Kommunikation mit dem Benutzer zu erlauben, sind in SALIERI rudimentäre Ein- und Ausgabeoperationen implementiert. Des weiteren stehen zwei Ausgabemöglichenkeiten für Notensegmente zur Verfügung.

**Ein- und Ausgabeoperationen**

| | |
|---|---|
| `read(`*string*`)` | liefert das Objekt, welches der Benutzer nach der Eingabeaufforderung *string* eingibt. |
| `readStr(`*string*`)` | liefert die Zeichenkette, welche der Benutzer nach der Eingabeaufforderung *string* eingibt. |
| `write(`$obj_1$`, `$obj_2$`, ..., `$obj_n$`)` | gibt $obj_1$ bis $obj_n$ ohne Zwischenraum auf dem Bildschirm aus. |
| `saveAsMidi(`*string*`, `*segm*`, `$int_1$`, `$int_2$`)` | Das Segment *segm* wird in der Datei *string* in MIDI-Format abgelegt. Optional können über $int_1$ und $int_1$ Zähler und Nenner des gewüschten Taktes angegeben werden. |
| `midiPlay(`*segm*`)` | Das Segment *segm* wird abgespielt. |

Mit der Funktion `read` können Objekte beliebigen Typs eingelesen werden. Dazu interpretiert das SALIERI-System die Eingabe als Ausdruck und ordnet dem Ergebnis den entsprechenden Typ zu. Will man die Eingaben des Benutzers selbst auswerten, gibt einem die Funktion `readStr` die Möglichkeit, unbearbeitete Zeichenreihen einzulesen.

**Beachte:**

- Soll die Auswertung der Eingabe verhindert werden, muß die Funktion `readStr` verwendet werden.
- Falls man bei der Funktion `read` eine Zeichenkette eingeben möchte, muß diese in Anführungszeichen gesetzt werden. Diese Notwendigkeit entfällt bei der Funktion `readStr`.
- Die Funktion `write` gibt ihre Parameter ohne Zwischenraum aus.

Die Funktion `saveAsMidi` schreibt das angegebene Segment in eine MIDI-Datei, welche später abgespielt werden kann. Für den interaktiven Betrieb bietet das SALIERI-System aber auch die Möglichkeit, sich das Segment mit der Funktion `midiPlay` sofort anzuhören.

**Beachte:**

- Durch die implizite Typkonvertierung des SALIERI-Systems können nicht nur Notensegmente, sondern auch Notensequenzen in MIDI-Dateien abgespeichert oder abgespielt werden.
- Das SALIERI-System hängt die Endung `.mid` automatisch (falls keine andere Extension angegeben wurde) an den Dateinamen an.

**Beispiel:** Ein- und Ausgabefunktionen

```
>read("Eingabe:  ")
Eingabe: 12
Result= 12
>read("Eingabe:  ")
Eingabe: 5<7
Result= TRUE
>read("Eingabe:  ")
Eingabe: abcd
Error in command-line: Unknown identifier: abcd
(occured while executing function "read")
>read("Eingabe:  ")
Eingabe: "abcd"
Result= "abcd"
>readStr("Eingabe:  ")
Eingabe: abcd
Result= "abcd"
>readStr("Eingabe:  ")
Eingabe: 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
>
```

*Bemerkung:* Die Funktion `midiPlay` erzeugt zwei MIDI-Dateien, wobei die zweite Datei (mit der Endung `.ply`) dem Wiedergabedämon signalisiert, daß eine MIDI-Datei zum Abspielen bereit liegt.

# 11 Sonstige Systemkommandos

Außer den in den vorhergehenden Abschnitten eingeführten Systemkommandos (vergl. Abschnitte 4, 3) stehen noch zwei weitere Kommandos zur Verfügung: `execDOS` und `dir`.

Das Kommando `execDOS` dient zur Ausführung von DOS-Befehlen und kann in zwei Varianten benutzt werden. In der Form `execDOS()` wird eine DOS-Shell aufgerufen, die die Eingabe und Ausführung beliebiger DOS-Kommandos erlaubt; das SALIERI-System bleibt dabei resident im Hintergrund. Durch Eingabe des DOS-Befehls `EXIT` kann man die DOS-Shell beenden und in das SALIERI-System zurückkehren. Mithilfe der alternativen Aufrufform `execDOS(`*string*`)` kann eine einzelne DOS-Kommandozeile *string* ausgeführt werden (diese ist als Zeichenkette anzugeben, vergl. Beispiel).

Weiterhin ist das Kommando `dir(`*maskStr*`)` verfügbar; dieses gibt die Namen aller Dateien im aktuellen Arbeitsverzeichnis aus, auf welche die Schablone *maskStr* paßt (vergl. Abschnitt 4).

**Beispiel:** Sonstige Systemkommandos

```
>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
>
```

**Beachte:**

- Um Datenverlust bei Programm- oder Systemabstürzen zu vermeiden, sollte vor dem Aufruf von DOS-Kommandos oder einer DOS-Shell der Workspace mit `sync()` gespeichert werden.
- Um eine DOS-Shell zu verlassen, muß `EXIT` (nicht etwa `exit()` eingegeben werden.

# 12 Fehlerbehandlung

Im interaktiven Betrieb wird es öfter einmal zu Mißverständnissen zwischen dem Benutzer und dem Rechner kommen, welche dieser anmaßenderweise als Fehler von Seiten des Benutzers interpretiert, wie wir das von Computern nicht anders gewöhnt sind.

Häufig handelt es sich dabei um einen falschen Funktionsaufruf, wobei der Funktionsname falsch eingegeben wurde, oder Zahl bzw. Typ der Parameter stimmen nicht mit den Erwartungen des SALIERI-Systems überein.

**Beispiel:** Falscher Funktionsaufruf

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

Da SALIERI eine funktionale Programmiersprache unterstützt, handelt es sich damit auch die Fehlerquelle Nr.1 der funktionalen Programmiersprachen ein: falsche Klammerstrukturen. Nun besitzt SALIERI auch noch eine Vielzahl verschieden geklammerter Konstrukte: Zeichenketten, Notensequenzen, Notensegmente, Blöcke, geklammerte Ausdrücke und geklammerte Parameterlisten von Funktionsaufrufen.

Fehlende runde Klammern führen meist zu einer allgemeinen Fehlermeldung, bei anderen Klammerzeichen passieren manchmal wunderliche Dinge.

**Beispiel:** Falsche Klammerstruktur

```
>(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]
>
```

Falls sich eine fehlerbehaftete Anweisung in einer benutzerdefinierten Funktion befindet, so wirkt sich der Fehler erst bei der Ausführung der Funktion aus. Dies soll das folgenden Beispiel verdeutlichen, wobei hier der häufige Fall des vergessenen Semikolons dargestellt wird.

**Beispiel:** Beispieldatei `error.sal`

```
f := FUNC(0,
      (* Fehlendes Semikolon in Zeile 4 *)
      ‘write("J.");
      write("S.")
      write("Bach");
');
```

**Beispiel:** Vergessenes Semikolon in einer benutzerdefinierter Funktion

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

Das Fehlen des Semikolons wird i.a. erst in der nächsten Zeile erkannt, wodurch die Zeilennummer ungenau wird. Die Angabe der Zeilennummer in einer benutzerdefinierten Funktion bezieht sich auf den Funktionskörper (d.h. die Zählung beginnt in der Zeile, in der das öffnende "`" steht).

**Beachte:**

Da ein Systemabsturz nicht ausgeschlossen werden kann, ist es sinnvoll, mit Hilfe des `sync`-Befehls von Zeit zu Zeit die aktuellen Daten im persistenten Workspace zu sichern.

# Index


`#`, 14
`!=`, 15, 17, 19, 29, 31, 32, 37
`*`, 11, 12
`+`, 11, 12, 14, 19, 21, 36
`-`, 11, 12
`/`, 12
`<=`, 15
`<`, 15
`=`, 15, 17, 19, 29, 31, 32, 37
`>=`, 15
`>`, 15
`@`, 36
`#`, 19, 21, 36

`abs`, 11, 12
`accel`, 20, 21
Akkordsequenz, 18
`ambitus`, 22
`and`, 11
Anweisungsblock, 32
`apply`, 30
arithmetische Operationen, 11
Aufruf, 5
Ausdrücke, 43
Ausdruck, 5, 8, 30
Ausdrucksblock, 32

Bibliothek, 38
Bibliotheken, 7
Blöcke, 32, 43
`bool` (Typ), 11
Boolesche Operationen, 11

`ceil`, 12
`comb`, 27
`cond`, 36
`consList`, 36
`cresc`, 20, 21

Datenstrukturen, 36
Datenstrukturen, benutzerdefiniert, 37
Datentypen, 11
Dauer, absolute, 22
Dauer, relative, 22
`del`, 14, 19, 36
`delete`, 9
`deleteAll`, 9
Deskriptorfunktionen, musikalische, 22
`dir`, 41
`div`, 11
DOS-Kommandos, Ausführen von, 41
DOS-Shell, 41
`doUntil`, 35
`duration`, 22

Ein- und Ausgabefunktionen, 39
einfach polyphone Form, 18
einfach polyphone Normalform, 18
`execDOS`, 41
`exists`, 10
`exit`, 4, 10
`expand`, 19, 21
Expansion, 19

`FALSE`, 11
Fehlerbehandlung, 42
`floor`, 12
`fragm`, 26, 27
`FUNC`, 28
Funktionen, benutzerdefiniert, 28, 43
Funktionen, intern, 28, 32
Funktionsaufruf, 28, 29, 42, 43
Funktionsdefinition, 28
Funktionsergebnis, 28
Funktionskörper, 28, 44
Funktionsparameter, 28, 31

`getInstr`, 22
`getIntens`, 22
`getList`, 36
`getPitch`, 22
`getTempo`, 22
Gleitpunktzahlen, 12

`if`, 33
`infix`, 14, 19, 36
`inList`, 36
`ins`, 14, 19, 36
`instr`, 20, 21
`int` (Typ), 11
Integer, 11
Integeroperationen, 11
Intensität, 16, 20, 22
interaktiver Betrieb, 40
Intervall, 23
`intvExpand`, 23

`invert`, 23
`isRest`, 22
`it`, 5

Klammerstruktur, 43
Klangfarbe, 16, 19, 20, 22
Kombinationsfunktionen, musikalische, 27
Kommando, 5
Kommandozeileneingabe, 5
Konkatenation, 14, 36
Kontrollstrukturen, 32

`length`, 14, 19, 36
`LIST`, 36
`list` (Typ), 36
Listen, 36, 43
Listenoperationen, 36
`load`, 10
`loop`, 33, 34
`loopn`, 33, 34

`MACRO`, 30
Makros, 28, 30
`merge`, 27
Metrum, 16, 20, 22
MIDI-Ausgabe, 19, 39
MIDI-Dateien, Erzeugen von, 39
MIDI-Dateien, Wiedergabe von, 39
MIDI-Kanäle, 19
`midiPlay`, 39
`mod`, 11

`not`, 11
Notensegment, 21, 26
Notensegmente, 16, 43
Notensequenzen, 16, 43

Objekt, 7
  erzeugen, 8
  inspizieren, 10
  löschen, 9
  Name, 7
  Typ, 7
  zuweisen, 8
`or`, 11

`part`, 21
`parts`, 21
Pause, 16, 21, 22, 24
`poly`, 21
`posNext`, 14, 19, 36
Programmaufruf, 5
Programmdateien, 6, 28
Protokolldatei, 7
`putList`, 36

`read`, 39
`readStr`, 39
`real` (Typ), 12
Realoperationen, 12
`RECORD`, Bibliothek, 38
record-Struktur, dynamische, 38
Reihenoperationen, 14, 19, 36
Rekursion, 28
`retro`, 19
`return`, 28
`rndUniform`, 11

*SALDIR*, 4
*SALPLYDIR*, 4
*SALTMPDIR*, 4
`save`, 10
`saveAsMidi`, 39
Schablone, 9
Schleifen, 33
`segment` (Typ), 16, 18
Segmentkonstanten, 18
Segmentoperationen, 21
`sequence` (Typ), 16
Sequenzkonstanten, 16
Sequenzoperationen, 19
`stat`, 10
`statAll`, 10
Stimmen, 21
`string` (Typ), 13
Stringoperationen, 14
`sync`, 10, 42, 44
Systemanforderungen, 3
Systemdateien, 4
Systemkommandos, 30, 32, 41

`time`, 22
Tonbeschreibung, 16
Tondauer, 16, 20
Tonhöhe, 16, 20, 22, 23
Tonleiter, 24
Tonparameter, 16
Tonumfang, 22
`transcr`, 24
Transkription, 24
`transp`, 19, 21, 24

Transposition, 19, 24
`transpTonal`, 24
`TRUE`, 11
`type`, 10
Typkonvertierung, 12, 40
Typprüfung, 28

Umgebungsvariable, 4
`until`, 35

Variationsfunktionen, 19, 20, 23, 26
`ver`, 5
Vergleichsoperationen, 15, 17, 19, 29, 31, 32, 37
Verzweigung, 33

Wahrheitswerte, 11
Wertzuweisung, 8, 28
`while`, 33, 34
Workspace, 7, 28, 30, 42, 44
    Persistenz, 10
    speichern, laden, 10
`write`, 10, 39
`writeAll`, 10

Zeichenketten, 13, 43