scogen 1.0

scogen ist ein Partitursynthese-Programm zur Erstellungen von *.sco*-Dateien für *Csound*. Es ist in *Scheme* programmiert und beinhaltet zudem eine ganze Reihe an hilfreichen Prozeduren. **scogen** ist beliebig erweiter- und veränderbar.

1. Installation

Voraussetzung für die Verwendung von **scogen** ist der Interpreter *guile* für *Scheme*.
Empfehlenswert ist die Verwendung mit dem Editor *Emacs* bzw. *Aquamacs*, für den es zudem eine **scogen**-Erweiterung des *scheme-mode* gibt.

1. Lade den Ordner *scogen-1.0* herunter und speicher ihn auf deinem Rechner.

2. *guile* speichert auf deinem Rechner die Initialisierungsdatei *boot-9.scm* - üblicherweise im Ordner *ice-9*.
Füge in die Datei *boot-9.scm* die folgende Zeile ein:

(load "/Users/name/scogen-1.0/scogen.scm")

Achte hierbei darauf, dass der Speicherpfad korrekt ist!
Wenn der **scogen**-mode nicht verwendet werden soll, funktioniert **scogen** nun bereits mit einer *.scm*-Datei.

3. Um den **scogen**-mode zu aktivieren, kopiere den gesamten Inhalt der Datei *scogen_mode* in die *.emacs*-Datei. Beim Öffnen einer *.sy*-Datei werden dadurch bestimmte Ausdrücke farblich gekennzeichnet und nach bestimmten Kriterien unterschieden.

2. Überblick: Funktionsweise

Die Syntax von **scogen** ist sehr einfach gehalten und benötigt in jedem Fall die Ausdrücke *scogen, lines!* und *parameters*. Der Ausdruck *tables* ist optional. An dieser Stelle ein sehr einfaches Beispiel:

```
(scogen "/Users/name/01.sco"
        (tables
         f# 1 0 4096 10 1)
        (lines! 4)
        (parameters
          '(1)
          (list 0 1 2 3)
          '(3)))
```

Beim Evaluieren dieses Ausdrucks wird die Datei *"/Users/name/scorefiles/01.sco"* erzeugt. Darauf folgt optional die Generierung von Funktionstabellen (mit einer leere Liste werden keine Tabellen erzeugt). Die Anzahl der Zeilen wird mit dem Ausdruck *lines!* definiert, der dadurch

automatisch die Zeilenanzahl an die Variable *lines* bindet. Auf den Ausdruck *parameters* folgen Listen, die die Parameter erzeugen. Wichtig hierbei ist, dass die einzelnen Parameter Listen sind. Die Anzahl der Parameter in der *.sco*-Datei wird dadurch automatisch bestimmt. Die mit diesem Beispiel erzeugte *.sco*-Datei lautet:

```
f 1 0 4096 10 1

i 1 0 3
i 1 1 3
i 1 2 3
i 1 3 3
```

Bei **scogen** ist es nicht relevant, wie lang die Listen für die Parameter sind. Bei jeder neuen Score-Zeile wird das jeweils nächste Element der Liste verwendet. Ist die Liste kürzer, als die Anzahl der Score-Zeilen, so fängt **scogen** wieder beim ersten Element der Liste an.

3. Syntax

scogen benötigt in jedem Fall den Ausruck *scogen* mit seiner Syntax:

(scogen <dateipfad>
<f-tables>
<zeilenanzahl>
<p-felder-liste>)

<dateipfad>: An dieser Stelle erwartet **scogen** den Pfad und Namen als String, an dem die *.sco*-Datei erstellt und gespeichert werden soll. Bsp.:

```
"/Users/name/01.sco"
```

<f-tables>: Sollen in die *.sco*-Datei Funktionstabellen geschrieben werden, lässt sich dies mit folgender Syntax programmieren:

(tables
<table 1>
<table 2>
. . .)

Die einzelnen Funktionstabellen werden nicht als Listen geschrieben. Das Beispiel

```
(tables
 f# 1 0 4096 10 1)
```

wird in der *.sco*-Datei zu:

```
f 1 0 4096 10 1
```

Soll eine Tabelle durch eine Hilfsprozedur erstellt werden, kann im **scogen**-Code auch eine Liste stehen,

die dann ausgewertet wird. Das Beispiel

```
(tables
 f# 1 0 4096 10 (lini 20 2 5))
```

wird in der *.sco*-Datei zu:

```
f 1 0 4096 10 20.0 15.5 11.0 6.5 2.0
```

(Die Prozedur *lini* erzeugt eine Liste mit 5 Werten, die zwischen 20 und 2 linear interpolieren)
Sollen mehrere Funktionstabellen erzeugt werden, muss jede neue Zeile mit dem Ausdruck *f#* beginnen, der dann automatisch eine neue Tabelle definiert.
Sollen keine Funktionstabellen erzeugt werden, muss an der Stelle <f-tables> eine leere Liste stehen.
Bsp.:

```
(scogen "/Users/name/01.sco"
        '()
        (lines! 4)
        (parameters
          '(1)
          (list 0 1 2 3)
          '(3)))
```

<zeilenanzahl>: Die Syntax für die Festlegung der Zeilenanzahl lautet:

(lines! <n>)

<n> muss eine ganze Zahl sein!

<p-felder-liste>: Syntax:

(parameters
 <p1>
 <p2>
 . . .)

Die Parameterfelder werden durch den Ausdruck *parameters* definiert. Jedes P-Feld muss eine Liste sein.

4. Integrierte Prozeduren

Der hilfreichste Teil von **scogen** sind die integrierten Prozeduren zum Generieren von Listen für die Parameter-Felder oder Funktionstabellen. Mit dem scheme-Ausdruck (scogen-manual) werden alle Prozeduren mit ihren benötigten Argumenten aufgelistet.
Unterschieden werden hierbei:
mathematische Funktionen, Listen erzeugende Prozeduren, Listen verändernde Prozeduren und Tonhöhenberechnungen.

mathematische Funktionen:

(roundx *n nachkommastellen*)

rundet die Zahl n auf die angegebene Anzahl Nachkommstellen
(roundx 0.1234 2) -> 0.12

(root *x n*)

zieht die x-te Wurzel aus n
(root 4 625) -> 5.0

(divs *n*)

schreibt alle Teiler von n in eine Liste
(divs 12) -> (1 2 3 4 6 12)

Listen erzeugende Prozeduren:

(lini *start ende anzahl*)

lineare Interpolation zwischen 2 Werten in einer bestimmten Anzahl
(lini 1 2 5) -> (1.0 1.25 1.5 1.75 2.0)

(linitol *start ende anzahl toleranz*)

lini mit Zufalls-Abweichungungen in festgelegter Toleranz (auch negativ!)
(linitol 1 2 5 2) -> Bsp.: (1 1.323 1.501 2.411 2)
(linitol 1 2 5 50) -> Bsp.: (1 24.115 -1.174 0.371 2)

(expi *start ende anzahl kruemmung*)

exponentielle Interpolation zwischen 2 Werten in einer bestimmten Anzahl unter Angabe der Kurvenkrümmung (nur Werte >= 0 zulässig!)
(expi 0 10 5 4) -> (0 0.039 0.625 3.164 10)

(expitol *start ende anzahl kruemmung toleranz*)

expi mit Zufalls-Abweichungungen in festgelegter Toleranz (auch negativ!)
(expitol 10 0 5 4 2) -> Bsp.: (10 3.702 1.027 -0.146 0)
(expitol 10 0 5 4 50) -> Bsp.: (10 12.67 -20.033 -11.088 0)

(dist *start abstand anzahl*)

erzeugt eine Liste mit gegebenem Startwert und gleichem Abstand zwischen den Werten mit einer bestimmten anzahl
(dist 0 1 4) -> (0.0 1.0 2.0 3.0)
(dist 2 -1.5 3) -> (2.0 0.5 -1)

(fibonacci *start anzahl*)

erzeugt eine Liste mit einer bestimmten Anzahl an Fibonacci-Zahlen von einem gegebenen Start-Index aus
(fibonacci 1 5) -> (0 1 1 2 3)
(fibonacci 2 5) -> (1 1 2 3 5)

(tend *unten-start unten-ende oben-start oben-ende anzahl*)

erzeugt eine Tendenzmaske (Zufallswerte) als Liste in gegebenem Rahmen
(tend -20 0 20 0 10) -> Bsp.: (8.133 -11.307 14.774 -9.372 9.908 -4.517 -2.201 -3.993 1.342 0.0)

(rand *minimum maximum anzahl*)

erzeugt eine Liste mit einer bestimmten Anzahl an Zufallswerten zwischen 2 Werten
(rand 30 33 5) -> Bsp.: (31.571 32.032 32.779 31.102 31.36)

Listen verändernde Prozeduren:

(sublist *liste start ende*)

zeigt nur einen Teil einer Liste
(sublist '(a b c d e f) 2 4) -> (c d e)

(fromto *liste from to anzahl ueberlappung*)

schafft einen Zufalls-Übergang zwischen 2 Elementen einer Liste mit einem gegebenen Überlappungsfaktor
(fromto '(a b c d e) 1 3 10 1) -> Bsp.: (b b b c c c c d d d)
(fromto '(a b c d e) 1 3 10 3) -> Bsp.: (b b b c b d d b d d)

(perm *liste laenge*)

zufällige Permutation einer Liste mit Angabe der neuen Länge
(perm '(1 2 3 4) 6) -> Bsp.: (3 3 2 4 2 4)

(procmap *liste proc wert*)

addiert/subtrahiert/multipliziert/dividiert jedes Element der Liste mit einem angegebenem Wert
(procmap '(1 2 3) * 10) -> (10 20 30)

(rep *liste n*)

wiederholt jedes Element der Liste n-mal
(rep '(a b c) 2) -> (a a b b c c)

(rep-rand *liste n-min n-max*)

wiederholt jedes Element der Liste zufällig oft zwischen n-min und n-max
(rep-rand '(a b c) 2 5) -> (a a a b b c c c c c)

(min *liste*)

schreibt den kleinsten Wert der Liste in eine Liste
(min '(5 0 3)) -> (0)

(max *liste*)

schreibt den größten Wert der Liste in eine Liste
(min '(5 0 3)) -> (5)

(min+max *liste*)

schreibt den kleinsten und größten Wert der Liste in eine Liste
(min '(5 0 3)) -> (0 5)

(mean *liste*)

schreibt den Mittelwert aller Zahlen einer Liste in eine Liste
(mean '(1 5 2 12)) -> (5.0)

(sort *liste*)

sortiert eine Liste an Zahlen von klein nach groß
(sort '(2 4 6 3 6)) -> (2 3 4 6 6)

(mix *listenliste*)

mischt alle Listen in einer Liste miteinander. Die Listen müssen gleich lang sein!
(mix '((1 4 7) (2 5 8) (3 6 9))) -> (1 2 3 4 5 6 7 8 9)

Tonhöhenberechnungen:

concert-pitch = 440

die Prozedur freq->pitch verwendet den Ausdruck concert-pitch. Soll er beispielsweise 443 Hz betragen, evaluiere den folgenden Ausdruck:
(set! concert-pitch 443)

(freq->pitch *frequenz*)

liefert eine Liste mit der Angabe der Tonhöhe in Pitch-Class-Notation und der Centabweichung
(freq->pitch 440) -> (8.09 0.0)
(freq->pitch 441) -> (8.09 3.93)

(harmonics *frequenz anzahl*)

erzeugt eine Liste mit eine bestimmten Anzahl an Teiltönen über einer gegebenen Frequenz
(harmonics 35.2 4) -> (35.2 70.4 105.6 140.8)

Anwendungsbeispiel mit einigen integrierten Prozeduren:

```
(scogen "/Users/name/01.sco"
        (tables
         f# 1 0 4096 10 1
         f# 2 0 4096 10 (expi 10 1 8 12))
        (lines! 12)
        (let ((p5 (tend 0 20 100 20 lines)))
          (parameters
           (rep '(1 2 3) 4)
           (dist 0 1 (/ lines 3))
           (rand 2 3 lines)
           (harmonics 35 (/ lines 4))
           p5
           (procmap p5 / 10)
           (divs 214)
           (append (tend 0 10 20 10 (/ lines 2)) (rep '(10) (/ lines 2))))))
```

daraus resultierende *.sco*-Datei:

```
f 1 0 4096 10 1
f 2 0 4096 10 10 7.385 5.412 3.934 2.834 2.021 1.429 1

i 1 0.0 2.701 35.0 24.891 2.4891 1 8.15
i 1 1.0 2.769 70.0 50.611 5.0611 2 12.317
i 1 2.0 2.333 105.0 44.583 4.4583 107 12.918
i 1 3.0 2.734 35.0 65.851 6.5851 214 11.521
i 2 0.0 2.621 70.0 66.861 6.6861 1 8.347
i 2 1.0 2.173 105.0 44.94 4.494 2 10.0
i 2 2.0 2.596 35.0 51.195 5.1195 107 10
i 2 3.0 2.207 70.0 43.356 4.3356 214 10
i 3 0.0 2.103 105.0 32.87 3.287 1 10
i 3 1.0 2.554 35.0 27.95 2.795 2 10
i 3 2.0 2.472 70.0 25.458 2.5458 107 10
i 3 3.0 2.447 105.0 20.0 2.0 214 10
```