

```

module Main where
import System.Random

-----
-- Project 1, a Haskell version of the PR-1 program of Gottfried Michael Koenig
-- interpretation and programming: W.G. Vree, 2007
-----
-- For each "musical parameter" this program calculates a sequence of Parts.
-- The musical parameters are:
-- chords, duration (called entry-delay) dynamics and tempo.
-- A Part can be either a Row a Group or a Balance.
-- Rows and Groups are sequences that obey certain rules of serial music.
-- A Balance is a (balanced) mixture of Rows and Groups.
-----

data Part a = Row [a] | Grp [a] | Bal [Part a] [Part a] | Par a

major :: [[Int]] -- explicit typing to force 32-bit integers (Int)
minor :: [[Int]] -- Haskell defaults to huge integers (Integer)
mapping :: [(String, Int, Int)]

-----
-- LEVEL 1 (parameter definitions, to be choosen by the composer)-----
-----

notes = ["c", "c#", "d", "d#", "e", "f", "f#", "g", "g#", "a", "a#", "b"]

major = [[0, 1, 5], [3, 4, 8], [6, 7, 11], [9, 10, 2]] -- interval-rows for computing chords
minor = [[0, 1, 4], [2, 3, 6], [5, 8, 9], [7, 10, 11]]

-- the possible entry-delays (chord duration) with corresponding minimum and maximum chord size

mapping = [(("1/1", 1, 6), ("4/5", 1, 6), ("3/4", 1, 6), ("2/3", 1, 6), ("5/8", 1, 6), ("3/5", 1, 6),
           ("1/2", 1, 4), ("2/5", 1, 4), ("3/8", 1, 4), ("1/3", 1, 4),
           ("1/4", 1, 3), ("1/5", 1, 3), ("1/8", 1, 2), ("0/0", 1, 1),
           ("1/2", 1, 4), ("2/5", 1, 4), ("3/8", 1, 4), ("1/3", 1, 4),
           ("1/4", 1, 3), ("1/5", 1, 3), ("1/8", 1, 2), ("0/0", 1, 1),
           ("1/4", 1, 3), ("1/5", 1, 3), ("1/8", 1, 2), ("0/0", 1, 1),
           ("1/8", 1, 2), ("0/0", 1, 1) ]

entry_list = map f mapping where f (x,y,z) = x -- extracted list of possible chord durations

dyna_list = ["ppp", "pp", "p", "mp", "mf", "f", "ff", "fff"] -- possible dynamics

tempo_list = ["t60", "t52", "t45.5", "t39.5", "t34.5", "t30"] -- possible tempo values (t60 == 60 1/4-beats per minute)

-- the following process-numbers specify the type of serial generation process (see: Level 4)
-- 1..3 == Rows are generated
-- 4 == Balance structures are generated
-- 5..7 == Groups are generated

dyna_process = 4::Int
entry_process = 4::Int
chord_process = 4::Int

rR = 1::Int -- 1..2, repetition rate for chord rows and groups

-- the average number of notes in a chord, caculated from the mapping above

average_chord_size = fromIntegral total / fromIntegral (2 * (length mapping))
  where
    total = sum [min + max | (delay, min, max) <- mapping]

-----
-- LEVEL 1 Section definition -----
-----

section nlines = output dyna_str entry_str tempo_str chord_str nlines
  where
    dyna_str = d_section dynamics
              (mk_row_str dyna_list rs1, mk_serial_str dyna_list rs2)
              dyna_process
              (length dyna_list)
              rs3
    where
      rs1 = mkRndStr 11
      rs2 = mkRndStr 12
      rs3 = mkRndStr 13

    entry_str = d_section entry_delay
              (mk_row_str entry_list rs1, mk_serial_str entry_list rs2)
              entry_process
              (length entry_list)
              rs3
    where
      rs1 = mkRndStr 14
      rs2 = mkRndStr 15
      rs3 = mkRndStr 16

    ch_size_str = d_section chord_size
              (flatten entry_str, [])
              0

```

```

        mapping
        rs1
    where
        rs1 = mkRndStr 17

tempo_str = d_section tempo_grp
    (flat2 entry_str, mk_serial_str tempo_list rs1)
    0
    (between 1 4 rl)
    []

    where
        (rl:rs1) = mkRndStr 18

chord_str = d_section chord
    (ch_size_str, mk_serial_str notes rs1)
    chord_process
    average_chord_size
    rs2

    where
        rs1 = mkRndStr 19
        rs2 = mkRndStr 20

-----
-- LEVEL 2 --General stream creation patterns-----
-----

d_section par_function streams process arguments rs =
    part : d_section par_function rest_streams process arguments rs1
    where
        ((part, rest_streams), rs1) = par_function streams process arguments rs

mk_row_str par_list rs = res1 : mk_row_str par_list rs1
    where
        (res1, rs1) = perm par_list rs

mk_serial_str par_list rs = foldr1 (++) (mk_row_str par_list rs)

-----
-- LEVEL 3 --Definition of the parameter functions-----
-----

dynamics str_tup process par_len rs
    | process <= 3 = (row str_tup (round (max 1 rowMax)), rs)
    | process == 4 = balance str_tup 2 (min 8 par_len) 1 rs
    | process > 4 = (group str_tup (between grpMin grpMax rl), rs1)
    where
        rowMax = fromIntegral (par_len * (5 - process)) / (fromIntegral 4)
        minvec = [[4, 6, 8], [4, 6, 8]]
        maxvec = [[6, 9, 12], [10, 15, 20]]
        grpMin = minvec !! (rR - 1) !! (process - 5)
        grpMax = maxvec !! (rR - 1) !! (process - 5)
        (rl : rs1) = rs

entry_delay str_tup process par_len rs
    | process <= 3 = (row str_tup (round (max 1 rowMax)), rs)
    | process == 4 = balance str_tup 1 (quot par_len 2) (between 1 4 rl) rs1
    | process > 4 = (group str_tup (between grpMin grpMax rl), rs1)
    where
        rowMax = fromIntegral (par_len * (5 - process)) / (fromIntegral 4)
        minvec = [[2, 5, 8], [3, 7, 11]]
        maxvec = [[4, 8, 12], [6, 11, 16]]
        grpMin = minvec !! (rR - 1) !! (process - 5)
        grpMax = maxvec !! (rR - 1) !! (process - 5)
        (rl : rs1) = rs

chord_size (delay : rest_delays, ys) process mapping rs =
    ((Par (between min max rl), (rest_delays, ys)), rs1)
    where
        ranges = [(min, max) | (del, min, max) <- mapping, del == delay]
        min = minimum (map fst ranges)
        max = maximum (map snd ranges)
        (rl : rs1) = rs

tempo_grp str_tup process rtc rs = (tempo_grp' str_tup process rtc, rs)
tempo_grp' (e:entries, t:tempos) process rtc =
    if rtc == 0 then (Grp [], (e:entries, tempos))
    else (Grp (t_list ++ rest_t_list), rest_streams)
    where
        t_list = replicate (length e) t
        (Grp rest_t_list, rest_streams) = tempo_grp' (entries, t:tempos) process (rtc - 1)

chord str_tup process average_ch_size rs
    | process <= 3 = row_chord str_tup (12 - (3 * (process - 1))) rs
    | process == 4 = balance_chord str_tup average_ch_size rs
    | process > 4 = one_of_3 (grp_tones str_tup (between min_tone max_tone rl) rs1)
        (grp_chord str_tup (between min_group max_group rl) 3 rs1)
        (grp_chord str_tup (between min_dgroup max_dgroup rl) 6 rs1)
        rs2

    where min_tone = process - 3
          max_tone = min_tone * (rR + 1)

```

```

min_group = round ((2.0 * average_ch_size * fromIntegral (min_tone)) / 2.25)
max_group = min_group * (rR + 1)
min_dgroup = round ((average_ch_size * fromIntegral (min_tone)) / 2.25)
max_dgroup = min_dgroup * (rR + 1)
(r1:r2:rs1) = rs
-----
-- LEVEL 4: Row, Group and Balance of non-chord parameters-----
-----
row ((xs : row_str), serial_str) m = (Row (take m xs), (row_str, serial_str))
group (row_str, (x : serial_str)) m = (Grp (replicate m x), (row_str, serial_str))
balance str_tup min max repeat rs = ((Bal set_parts rs1, rest_streams2), rs3)
  where
    (len_list, rs1)           = mk_len_list min max repeat rs
    (proc_list, rs2)          = mk_proc_list      repeat rs1
    set_proc_list              = map fst proc_list
    bal_proc_list              = map snd proc_list
    (set_parts, rest_streams) = struc set_proc_list str_tup len_list
    (balance_parts, rest_streams2) = struc bal_proc_list rest_streams len_list
    (rs1, rs3)                = perm balance_parts rs2

    struc fs str_tup [] = ([], str_tup)
    struc (f:fs) str_tup (len:lens) = ((part : parts), rest_streams2)
      where
        (part, rest_streams) = f str_tup len
        (parts, rest_streams2) = struc fs rest_streams lens

    mk_len_list min max 0 rs = ([], rs)
    mk_len_list min max n rs = (between min max r1 : rs1, rs2)
      where
        (r1 : rs1) = rs
        (rs1, rs2) = mk_len_list min max (n - 1) rs1

    mk_proc_list 0 rs = ([], rs)
    mk_proc_list n rs = (one_of (row, group) (group, row) r1 : rs1, rs2)
      where
        (r1 : rs1) = rs
        (rs1, rs2) = mk_proc_list (n - 1) rs1
-----
-- LEVEL 4: row, group, double-group, tone and balance of the chord parameter
-----
row_chord (ch_len_str, note_str) row_size rs = ((Row chords, (rest_ch_len_str, rest_note_str)), rs1)
  where
    ((trio_notes, rest_note_str), rs1) = trio_row note_str rs
    row = take row_size trio_notes
    (chords, rest_ch_len_str) = fill_chord ch_len_str row

grp_chord (ch_len_str, note_str) ngroups group_size rs = ((Grp chords, (rest_ch_len_str, rest_note_str)), rs1)
  where
    ((trio_notes, rest_note_str), rs1) = trio_row note_str rs
    group = take group_size trio_notes
    groups = foldr1 (++) (replicate ngroups group)
    (chords, rest_ch_len_str) = fill_chord ch_len_str groups

grp_tones (ch_len_str, note : rest_note_str) ntones rs = ((Grp chords, (rest_ch_len_str, rest_note_str)), rs1)
  where
    first_notes = note : first_notes
    ((chords, rest_ch_len_str), rs1) = mk_tones ch_len_str first_notes ntones rs

mk_tones ch_len_str first_notes 0 rs = ([], ch_len_str, rs)
mk_tones (Par ch_len : ch_len_str) first_notes ntones rs = ((tone_sp : rest_tones, rest_ch_len_str), rs2)
  where
    ((trio notes, _), rs1) = trio_row first_notes rs
    chord = take ch_len trio_notes
    tone = head first_notes : tail chord
    tone_sp = foldr1 (\x y -> x ++ " " ++ y) tone
    ((rest_tones, rest_ch_len_str), rs2) = mk_tones ch_len_str first_notes (ntones - 1) rs1

balance_chord str_tup avrage_ch_size rs = one_of exp1 exp2 r1
  where
    exp1 = ((Bal [Row chords] [chord_grp], rest_str_tup2), rs3) where
      nrows = fromIntegral (between 1 3 r2)
      ntones = round ((nrows * 5.0 * 2.25) / avrage_ch_size)
      ngroups = round ((nrows * 4.0 * 2.25) / avrage_ch_size)
      ndgroups = round ((nrows * 2.0 * 2.25) / avrage_ch_size)
      ((chords, rest_str_tup1), rs2) = rep_chord_row str_tup nrows rs1
      ((chord_grp, rest_str_tup2), rs3) = one_of_3 (grp_tones rest_str_tup1 ntones rs2)
        (grp_chord rest_str_tup1 ngroups 3 rs2)
        (grp_chord rest_str_tup1 ndgroups 6 rs2)
        r2

    exp2 = ((Bal [chords_grp] [Row chords], rest_str_tup2), rs3) where
      minElem = one_of_3 6 4 2 r2
      maxElem = one_of_3 (18 * rR)
        (round ((12.0 * fromIntegral (rR) * avrage_ch_size) / 2.25))
        (round ((6.0 * fromIntegral (rR) * avrage_ch_size) / 2.25))
        r2

```

```

nelems = between minElem maxElem r2
nrows = round (fromIntegral (nelems) / (one_of_3 5.0 4.0 2.0 r2))
((chords_grp, rest_str_tup1), rs2) = one_of_3 (grp_tones str_tup nelems rs1)
                                     (grp_chord str_tup nelems 3 rs1)
                                     (grp_chord str_tup nelems 6 rs1)
                                     r2
((chords, rest_str_tup2) , rs3) = rep_chord_row rest_str_tup1 nrows rs2
(r1:r2:rs1) = rs

-----
-- support-functions for the chord parameter functions of level 4 -----
-----

trio_row (start_note : rest_note_str) rs = ((map (add_note start_note) trio_list, rest_note_str), rs2)
  where
    interval_row = one_of major minor r1
    mixed_row    = foldr1 (++) res1
    trio_list    = one_of mixed_row (reverse mixed_row) r2
    (res1, rs2)  = perm interval_row rs1
    (r1:r2:rs1) = rs

fill_chord ch_len_str row =
  if length row >= chlen then (chord_sp : rest_chords, rest_ch_len_str)
  else ([], ch_len_str)
  where (Par chlen : rest_lens) = ch_len_str
        (chord, rest_row)      = splitAt chlen row
        (rest_chords, rest_ch_len_str) = fill_chord rest_lens rest_row
        chord_sp = foldr1 (\x y -> x ++ " " ++ y) chord

rep_chord_row str_tup 0 rs = ([], str_tup)
rep_chord_row str_tup nrows rs = ((chords ++ rest_chords, rest_str_tup2), rs2)
  where
    ((Row chords, rest_str_tup1), rs1) = row_chord str_tup 12 rs
    ((rest_chords, rest_str_tup2), rs2) = rep_chord_row rest_str_tup1 (nrows - 1) rs1

-----
-- SMALL UTILITY FUNCTIONS -----
-----

select r xs = xs !! (mod r (length xs))

one_of x y r = select r [x,y]
one_of_3 x y z r = select r [x,y,z]

between left right r = left + mod r (1 + right - left)

perm [x] rs = ([x], rs)
perm xs (r:rs) = ((v:ys), rs1)
  where
    (us, (v:vs)) = splitAt (mod r (length xs)) xs
    (ys, rs1) = perm (us ++ vs) rs

mkRndStr n = rndStr (mkStdGen n)
  where
    rndStr rg = r1 : rndStr rg1
              where (r1,rg1) = next rg

add_note notel interval = notes !! iy
  where
    index note [] = -1
    index note (x:xs) | note == x = 0
                      | otherwise = 1 + index note xs
    ix = index notel notes
    iy = rem (ix + interval) 12

flatten [] = []
flatten (Row x : xs) = x ++ flatten xs
flatten (Grp x : xs) = x ++ flatten xs
flatten (Bal xs ys : zs) = flatten xs ++ flatten ys ++ flatten zs

flat2 [] = []
flat2 (Row x : xs) = x : flat2 xs -- row structure is kept
flat2 (Grp x : xs) = x : flat2 xs -- group structure is kept
flat2 (Bal xs ys : zs) = flat2 xs ++ flat2 ys ++ flat2 zs -- row/group structure is kept

-----
-- OUTPUT FUNCTIONS -----
-----
-- one section is output as a table. Before each parameter the start of a new
-- row or group is marked with 'r' or 'g'. In addition the start of a balance
-- structure is marked with an 's' (set) followed later by a 'b' (balance)
-----

showP x n = replicate (n - (length str)) ' ' ++ str where str = show x

tagRow n b (x:xs) = (b ++ "r " ++ showP x n) : tagRest n xs
tagGrp n b (x:xs) = (b ++ "g " ++ showP x n) : tagRest n xs
tagPrm n b x = ["p " ++ showP x n]
tagRest n [] = []
tagRest n (x:xs) = (" " ++ showP x n) : tagRest n xs

tagPar n b [] = []

```

```

tagPar n b (Par x      : xs) = tagPrm n b x ++ tagPar n " " xs
tagPar n b (Row x      : xs) = tagRow n b x ++ tagPar n " " xs
tagPar n b (Grp x      : xs) = tagGrp n b x ++ tagPar n " " xs
tagPar n b (Bal xs ys : zs) = tagPar n "s" xs ++ tagPar n "b" ys ++ tagPar n b zs

output dyna_str entry_str tempo_str chord_str nlines = pr_lines (take nlines lines)
  where
    lines = pr_tagstr (tagPar 5 " " dyna_str) (tagPar 5 " " entry_str) (tagPar 7 " " tempo_str) (tagPar 1 " " chord_str)
    pr_tagstr (x:xs) (y:ys) (z:zs) (u:us) = (x ++ " |" ++ y ++ " |" ++ z ++ " |" ++ u) : pr_tagstr xs ys zs us
    pr_lines [] = []
    pr_lines (x:xs) = x ++ "\n" ++ pr_lines xs

main = putStr (section 50) -- print one section of xx lines

```