DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Anexo código mini-collider

Teoría de lenguajes

Grupo: 11

Integrante	LU	Correo electrónico
Calderini, Nicolás	820/10	calderini.nicolas@gmail.com
Hernández, Santiago	48/11	santi-hernandez@hotmail.com
Marasca, Dardo	227/07	dmarasca@yahoo.com.ar
Saravia, Nicolás	905/04	nicolasaravia@yahoo.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

El código entregado se encuentra dividido en módulos con la siguiente estructura:

```
+--- mini_collider.py
+--- minicollider
|    \--- lexer.py
|    \--- mixer.py
|    \--- parser.py
|    \--- test
|         \--- test_mixer.py
|         \--- test_parser.py
```

1. mini_collider.py

```python
import minicollider.parser
try:
    import argparse
except ImportError:
    from minicollider.external import argparse


def parsear_argumentos():
    argparser = argparse.ArgumentParser(formatter_class=
                                    argparse.ArgumentDefaultsHelpFormatter)
    argparser.add_argument('-s', '--samplerate',
                           help="The desired sample rate.",
                           default=8000,
                           type=int)
    argparser.add_argument('-b', '--beat',
                           help="The desired beat.",
                           default=8000 / 12,
                           type=int)
    argparser.add_argument('-f', '--file',
                           help="A file with a buffer to parse (optional).")
    return argparser.parse_args()


def parsear_archivo(file):
    try:
        archivo = open(args.file, 'r')
        entrada = archivo.read()
        archivo.close()
    except IOError:
        print 'Error opening the file.'
        exit(1)
    try:
        minicollider.parser.parse(entrada)
    except Exception, e:
        if e: msg = str(e)
        else: msg = 'Syntax Error'
        print "Error: %s" % msg

def prompt():
    while 1:
        try:
            entrada = raw_input('buffer > ')
        except EOFError:
            print
            break
        if entrada != '':
            try:
                minicollider.parser.parse(entrada)
            except Exception, e:
                print "Error: %s" % e



if __name__ == '__main__':
    args = parsear_argumentos()
    minicollider.parser.init(args.samplerate, args.beat)
    if args.file is not None:
        parsear_archivo(args.file)
    else:
        prompt()
```

2. lexer.py

```
# ---------------------------------------------------------------------------
# lexer.py
#
# Lexer para el mini-collider
# ---------------------------------------------------------------------------
try:
    import ply.lex as lex
except ImportError:
    import external.lex as lex
import re

tokens = (
    'NUM', 'SIN', 'LIN', 'SIL', 'NOI', 'PLAY', 'POST', 'LOOP',
    'TUNE', 'FILL', 'REDU', 'EXPA', 'CON', 'MIX', 'ADD', 'SUB', 'MUL',
    'DIV', 'LPAREN', 'RPAREN', 'LLLAVE', 'RLLAVE','PLOT', 'COMA',
)

# Tokens

t_SIN          = r'sin'
t_LIN          = r'linear|lin'
t_SIL          = r'silence|sil'
t_NOI          = r'noise|noi'
t_PLAY         = r'.play'
t_POST         = r'.post'
t_LOOP         = r'.loop'
t_TUNE         = r'.tune'
t_FILL         = r'.fill'
t_REDU         = r'.reduce'
t_EXPA         = r'.expand'
t_PLOT         = r'.plot'
t_CON          = r'con|;'
t_MIX          = r'mix|&'
t_ADD          = r'add|\+'
t_SUB          = r'sub|-'
t_MUL          = r'mul|\*'
t_DIV          = r'div|/'
t_LPAREN       = r'\('
t_RPAREN       = r'\)'
t_LLLAVE       = r'\{'
t_RLLAVE       = r'\}'
t_COMA         = r','
t_ignore_COMM = r'//.*\n'
t_ignore_WS    = r'\s|\t|\n'

def t_NUM(t):
    r'\d+(\.\d+)?'
    if t.value.find('.') == -1:
        t.value = int(t.value)
    else:
        t.value = float(t.value)

    return t

def t_error(t):
    raise SyntaxError("Caracter ilegal: '%s'" % t.value[0])
```

```

# Build the lexer
lexer = lex.lex()
```

3. mixer.py

```python
import math
import numpy
import pylab
import pygame


NUMPY_ENCODING = numpy.int16
MIXER_ENCODING = -16

MAX_AMPLITUDE = 32000
SAMPLE_RATE = 8800
BEAT = SAMPLE_RATE / 12


def init(sample_rate=8800, beat=8800/12, init_pygame=1):
    global SAMPLE_RATE, BEAT

    SAMPLE_RATE = sample_rate
    BEAT = beat

    if init_pygame:
        pygame.mixer.pre_init(SAMPLE_RATE, MIXER_ENCODING, 1)
        pygame.init()


class Sound():
    def __init__(self, samples):
        if (len(samples)) == 0:
            raise Exception('No se puede crear un buffer vacio')

        self.samples = numpy.array(samples, numpy.float)

    def __eq__(self, other):
        return numpy.array_equal(self.samples, other.samples)

    def __iter__(self):
        return self.samples.__iter__()

    def __len__(self):
        return len(self.samples)

    def __add__(self, other):
        return self._oper(other, (lambda x, y: x + y))

    def __mul__(self, other):
        return self._oper(other, (lambda x, y: x * y))

    def __sub__(self, other):
        return self._oper(other, (lambda x, y: x - y))

    def __div__(self, other):
        return self._oper(other, (lambda x, y: x / y))

    def __floordiv__(self, other):
        return self.concat(other)

    def __and__(self, other):
        return self._oper(other, (lambda x, y: (x + y) / 2))

    def get_samples(self):
        return self.samples

    def set_samples(self, samples):
        self.samples = samples
        return self

    def play(self, speed):
        amp_mul = MAX_AMPLITUDE / numpy.amax(self.samples)
        samples = numpy.array(self.get_samples() * amp_mul, NUMPY_ENCODING)
        channel = pygame.sndarray.make_sound(samples).play()
        while channel.get_busy(): pass
        return self

    def plot(self):
        pylab.plot(numpy.arange(int(len(self.samples))), self.samples)
        pylab.show()
        return self

    def post(self):
        print self
        return self

    def loop(self, count):
        if not(0 < count):
            raise Exception("[LOOP] Se esperaba un numero positivo: %s" % count)
        return self.resize(int(count * len(self.samples)))

    def resize(self, new_len):
        if not(isinstance(new_len, int) and 0 < new_len):
            raise Exception("[RESIZE] Se esperaba un entero positivo: %s" % new_len)
        new_samples = numpy.zeros(new_len)
        for i in xrange(new_len):
            new_samples[i] = self.samples[i % len(self.samples)]
        return Sound(new_samples)

    def resample(self, new_len):
        if not(isinstance(new_len, int) and 0 < new_len):
            raise Exception("[RESAMPLE] Se esperaba un entero positivo: %s" % new_len)
        new_samples = numpy.zeros(new_len)
        for i in xrange(new_len):
            new_samples[i] = self.samples[int(i * len(self) / new_len)]
        return Sound(new_samples)

    def copy(self):
        return Sound(self.samples)

    def concat(self, other):
        new_samples = numpy.concatenate((self.samples, other.samples))
        return Sound(new_samples)

    def tune(self, pitch):
        return self.resample(int(
                len(self)
                * ( (2**(1.0/12))**(-pitch) )
        ))

    def fill(self, count):
        if not(0 < count):
            raise Exception("[FILL] Se esperaba un numero positivo: %s" % count)
        new_len = int(BEAT * count)

        if (len(self) >= new_len): return self.copy()

        new_samples = numpy.zeros(new_len)
        for i in xrange(len(self)):
            new_samples[i] = self.samples[i]
        return Sound(new_samples)

    def reduce(self, count=1):
        if not(0 < count):
            raise Exception("[REDUCE] Se esperaba un numero positivo: %s" % count)
        new_len = int(count * BEAT)
        if (len(self) > new_len):
            return self.resample(new_len)
        else:
            return self.copy()

    def _oper(self, other, op):
        if (len(self) < len(other)):
            a = self.resize(len(other))
            b = other
        else:
            a = self
            b = other.resize(len(self))

        new_samples = numpy.zeros(len(a))
        for i in xrange(len(a)):
            new_samples[i] = op(a.samples[i], b.samples[i])
        return Sound(new_samples)

    def expand(self, count=1):
        if not(0 < count):
            raise Exception("[EXPAND] Se esperaba un numero positivo: %s" % count)
        new_len = int(count * BEAT)
        if (len(self) < new_len):
            return self.resample(new_len)
        else:
            return self.copy()

    def __str__(self):
        return str(self.samples)

    def tolist(self):
        return self.samples.tolist()

class SoundGenerator():
    def __init__(self):
        pass

    def get_sample_rate(self):
        return SAMPLE_RATE

    def get_beat(self):
        return BEAT

    def get_beats_per_second(self):
        return SAMPLE_RATE / BEAT

    def from_list(self, samples):
        return Sound(numpy.array(samples))

    def sine(self, cicles, amp):
        if not(0<= amp <=1):
            raise Exception("[SINE] Amplitud incorrecta: %s" % amp)
        if not(0 < cicles and isinstance(cicles, int)):
            raise Exception("[SINE] Valor de ciclos incorrecto: %s" % cicles)
        omega = (cicles * numpy.pi * 2) / BEAT
        xvalues = numpy.arange(BEAT) * omega
        return Sound(amp * numpy.sin(xvalues))

    def silence(self):
        return Sound(numpy.zeros(BEAT))

    def linear(self, start, end):
        if not(-1<= start <=1 and -1<= end <=1):
            raise Exception("[LINEAR]  Rango incorrecto: %s, %s" % (start, end))
        return Sound(numpy.linspace(start, end, BEAT))

    def noise(self, amp):
        if not(0<= amp <=1):
            raise Exception("[NOISE] Amplitud incorrecta: %s" % amp)
        return Sound(numpy.random.random(BEAT) * amp)

    def note(self, note, amp, octave=1):
        freqencies = {
            'C' : 261.63,
            'D' : 293.66,
            'E' : 329.63,
            'F' : 349.23,
            'G' : 392,
            'A' : 440,
            'B' : 493.88
        }
        return self.sine(freqencies[note] * octave / self.get_beats_per_second(), amp)
```

4. parser.py

```python
# -----------------------------------------------------------------------------
# parser.py
#
# Parser para el mini-collider
# -----------------------------------------------------------------------------
try:
    import ply.yacc as yacc
except ImportError:
    import external.yacc as yacc
from lexer import tokens
import mixer

generator = None


def init(sample_rate, beat, init_pygame=1):
    global generator
    mixer.init(sample_rate, beat, init_pygame)
    generator = mixer.SoundGenerator()


def parse(input):
    try:
        res = parser.parse(input)
    except Exception, e:
        raise SyntaxError("%s" % e)
    return res

precedence = (
    ('left', 'CON', 'MIX'),
    ('nonassoc', 'LOOP', 'POST', 'TUNE', 'EXPA', 'REDU','FILL', 'PLAY', 'PLOT'),
    ('left', 'ADD', 'SUB'),
    ('left', 'MUL', 'DIV'),
)


def p_statement_expr(t):
    'START : BUFFER'

    t[0] = t[1]


def p_BUFFER_binop(t):
    '''BUFFER :       BUFFER CON BUFFER
                |     BUFFER MIX BUFFER
                |     BUFFER ADD BUFFER
                |     BUFFER SUB BUFFER
                |     BUFFER MUL BUFFER
                |     BUFFER DIV BUFFER'''

    if   t[2] in ['con', ';']: t[0] = t[1] // t[3]
    elif t[2] in ['mix', '&']: t[0] = t[1] & t[3]
    elif t[2] in ['add', '+']: t[0] = t[1] + t[3]
    elif t[2] in ['sub', '-']: t[0] = t[1] - t[3]
    elif t[2] in ['mul', '*']: t[0] = t[1] * t[3]
    elif t[2] in ['div', '/']: t[0] = t[1] / t[3]
```

```python


def p_BUFFER_metodo_0param(t):
    '''BUFFER :       BUFFER PLAY ONEPARAM
                |     BUFFER POST ONEPARAM
                |     BUFFER LOOP ONEPARAM
                |     BUFFER TUNE ONEPARAM
                |     BUFFER FILL ONEPARAM
                |     BUFFER REDU ONEPARAM
                |     BUFFER PLOT ONEPARAM
                |     BUFFER EXPA ONEPARAM '''

    try:
        if   t[2] == '.play': t[0] = t[1].play(1)
        elif t[2] == '.post': t[0] = t[1].post()
        elif t[2] == '.loop': t[0] = t[1].loop(1)
        elif t[2] == '.tune': t[0] = t[1].tune(1)
        elif t[2] == '.fill': t[0] = t[1].fill(1)
        elif t[2] == '.plot': t[0] = t[1].plot()
        elif t[2] == '.reduce': t[0] = t[1].reduce(1)
        elif t[2] == '.expand': t[0] = t[1].expand(1)
    except Exception, e:
        print "Syntax error: %s" % e
        raise SyntaxError


def p_ONEPARAM(t):
    '''ONEPARAM :       LPAREN RPAREN
                | '''


def p_BUFFER_metodo_1param(t):
    '''BUFFER :       BUFFER PLAY LPAREN NUM RPAREN
                |     BUFFER LOOP LPAREN NUM RPAREN
                |     BUFFER FILL LPAREN NUM RPAREN
                |     BUFFER REDU LPAREN NUM RPAREN
                |     BUFFER EXPA LPAREN NUM RPAREN '''

    if   t[2] == '.play': t[0] = t[1].play(t[4])
    elif t[2] == '.loop': t[0] = t[1].loop(t[4])
    elif t[2] == '.fill': t[0] = t[1].fill(t[4])
    elif t[2] == '.reduce': t[0] = t[1].reduce(t[4])
    elif t[2] == '.expand': t[0] = t[1].expand(t[4])


def p_BUFFER_metodo_1param_tune_pos(t):
    '''BUFFER :    BUFFER TUNE LPAREN NUM RPAREN'''
    t[0] = t[1].tune(t[4])


def p_BUFFER_metodo_1param_tune_neg(t):
    '''BUFFER :    BUFFER TUNE LPAREN SUB NUM RPAREN'''
    t[0] = t[1].tune(-t[5])


def p_BUFFER_generador_0param(t):
    '''BUFFER :     SIL ONEPARAM
                |   NOI ONEPARAM'''

    if   t[1] in ['silence','sil']: t[0] = generator.silence()
    elif t[1] in ['noise','noi']: t[0]   = generator.noise(1)


def p_BUFFER_generator_1param(t):
    '''BUFFER :       SIN LPAREN NUM RPAREN
                |     NOI LPAREN NUM RPAREN '''

    if   t[1] == 'sin': t[0] = generator.sine(t[3], 1)
    elif t[1] in ['noise','noi']: t[0] = generator.noise(t[3])


def p_BUFFER_generator_2param_sin(t):
    'BUFFER : SIN LPAREN NUM COMA NUM RPAREN '

    t[0] = generator.sine(t[3], t[5])


def p_BUFFER_generator_2param_lin_pos_pos(t):
    'BUFFER : LIN LPAREN NUM COMA NUM RPAREN'

    t[0] = generator.linear(t[3], t[5])


def p_BUFFER_generator_2param_lin_neg_pos(t):
    'BUFFER : LIN LPAREN SUB NUM COMA NUM RPAREN'

    t[0] = generator.linear(-t[4], t[6])


def p_BUFFER_generator_2param_lin_pos_neg(t):
    'BUFFER : LIN LPAREN NUM COMA SUB NUM RPAREN'

    t[0] = generator.linear(t[3], -t[6])


def p_BUFFER_generator_2param_lin_neg_neg(t):
    'BUFFER : LIN LPAREN SUB NUM COMA SUB NUM RPAREN'

    t[0] = generator.linear(-t[4], -t[7])


def p_BUFFER_llaves(t):
    'BUFFER : LLLAVE BUFFER RLLAVE '

    t[0] = t[2]


def p_minus_number(t):
    'BUFFER : SUB NUM'

    t[0] = generator.from_list([- t[2]])


def p_expression_number(t):
    'BUFFER : NUM'

    t[0] = generator.from_list([t[1]])


def p_error(t):
    raise SyntaxError("Syntax error in input!")

parser = yacc.yacc()
```

5. test_mixer.py

```
import minicollider.mixer as mixer
import numpy
import unittest

class MixerTestCase(unittest.TestCase):


    def assertElementsInRange(self, list, min, max):
        for item in list:
            self.assertTrue(min <= item <= max)

    def setUp(self):
        self.sample_rate = 4800
        self.beat = self.sample_rate / 12

        mixer.init(self.sample_rate, self.beat, 0)
        self.generator = mixer.SoundGenerator()


class TestSoundGeneratorCases(MixerTestCase):


    def test_from_list(self):

        sound = self.generator.from_list([1])
        self.assertEqual([1], sound.tolist())

        sound = self.generator.from_list([0.5])
        self.assertEqual([0.5], sound.tolist())

        sound = self.generator.from_list([0, 0.1, -1])
        self.assertEqual([0, 0.1, -1], sound.tolist())

        self.assertRaises(Exception, lambda : self.generator.from_list([]))


    def test_silence(self):
        sound = self.generator.silence()
        self.assertEqual([0] * self.beat, sound.tolist())


    def test_linear(self):
        sound = self.generator.linear(0, 0)
        self.assertEqual([0] * self.beat, sound.tolist())

        sound = self.generator.linear(1, 1)
        self.assertEqual([1] * self.beat, sound.tolist())

        sound = self.generator.linear(0.5, 0.5)
        self.assertEqual([0.5] * self.beat, sound.tolist())

        sound = self.generator.linear(-1, -1)
        self.assertEqual([-1] * self.beat, sound.tolist())

        sound = self.generator.linear(0, -1);
        self.assertTrue(numpy.array_equal(numpy.linspace(0, -1, self.beat),
                                            sound.get_samples()))
        self.assertEqual(self.beat, len(sound))

        sound = self.generator.linear(1, -1);
        self.assertTrue(numpy.array_equal(numpy.linspace(1, -1, self.beat),
                                            sound.get_samples()))
        self.assertEqual(self.beat, len(sound))

        sound = self.generator.linear(0, 1);
        self.assertTrue(numpy.array_equal(numpy.linspace(0, 1, self.beat),
                                            sound.get_samples()))
        self.assertEqual(self.beat, len(sound))

        sound = self.generator.linear(-1, 1);
        self.assertTrue(numpy.array_equal(numpy.linspace(-1, 1, self.beat),
                                            sound.get_samples()))
        self.assertEqual(self.beat, len(sound))

        self.assertRaises(Exception, lambda : self.generator.linear(2, 0))
        self.assertRaises(Exception, lambda : self.generator.linear(0, 2))


    def test_noise(self):
        sound = self.generator.noise(0)
        self.assertEqual([0] * self.beat, sound.tolist())

        sound = self.generator.noise(1)
        self.assertElementsInRange(sound, -1, 1)
        self.assertEqual(len(numpy.unique(sound.get_samples())), len(sound))
        self.assertEqual(self.beat, len(sound))

        sound = self.generator.noise(0.5)
        self.assertElementsInRange(sound, -0.5, 0.5)
        self.assertEqual(len(numpy.unique(sound.get_samples())), len(sound))
        self.assertEqual(self.beat, len(sound))

        sound = self.generator.noise(0.1)
        self.assertElementsInRange(sound, -0.1, 0.1)
        self.assertEqual(len(numpy.unique(sound.get_samples())), len(sound))
        self.assertEqual(self.beat, len(sound))


    def test_sine(self):
        sound = self.generator.sine(1, 0)
        self.assertEqual([0] * self.beat, sound.tolist())

        sound = self.generator.sine(1, 1)
        self.assertEqual(self.beat, len(sound))
        self.assertElementsInRange(sound, -1, 1)

        sound = self.generator.sine(4, 0.5)
        self.assertEqual(self.beat, len(sound))
        self.assertElementsInRange(sound, -0.5, 0.5)

        self.assertRaises(Exception, lambda : self.generator.sine(0, 1))
        self.assertRaises(Exception, lambda : self.generator.sine(-1, 1))
        self.assertRaises(Exception, lambda : self.generator.sine(0.5, 1))
        self.assertRaises(Exception, lambda : self.generator.sine(1, -1))
        self.assertRaises(Exception, lambda : self.generator.sine(0, 2))


class TestSoundCases(MixerTestCase):


    def test_add(self):

        sound1 = self.generator.from_list([0, 1, -0.5])
        sound2 = self.generator.from_list([0.5, -0.2, 0.5])
        sound3 = sound1 + sound2
        self.assertEqual([0.5, 0.8, 0], sound3.tolist())

        sound1 = self.generator.from_list([0.1])
        sound2 = self.generator.from_list([0, 0.4, 0.5])
        sound3 = sound1 + sound2
        self.assertEqual([0.1, 0.5, 0.6], sound3.tolist())
        sound4 = sound2 + sound1
        self.assertEqual(sound3, sound4)

        sound1 = self.generator.from_list([0.1, -0.1])
        sound2 = self.generator.from_list([0, 0.5, 0.5, 0.6])
        sound3 = sound1 + sound2
        self.assertEqual([0.1, 0.4, 0.6, 0.5], sound3.tolist())
        sound4 = sound2 + sound1
        self.assertEqual(sound3, sound4)


    def test_sub(self):

        sound1 = self.generator.from_list([0, 0, -0.5])
        sound2 = self.generator.from_list([0.5, -0.2, 0.5])
        sound3 = sound1 - sound2
        self.assertEqual([-0.5, 0.2, -1], sound3.tolist())

        sound1 = self.generator.from_list([0.1])
        sound2 = self.generator.from_list([0, 0.2, 0.5])
        sound3 = sound1 - sound2
        self.assertEqual([0.1, -0.1, -0.4], sound3.tolist())

        sound1 = self.generator.from_list([0.1])
        sound2 = self.generator.from_list([0, 0.2, 0.5])
        sound3 = sound2 - sound1
        self.assertEqual([-0.1, 0.1, 0.4], sound3.tolist())

        sound1 = self.generator.from_list([0.1, -0.1])
        sound2 = self.generator.from_list([0, 0.5, 0.5, 0.6])
        sound3 = sound1 - sound2
        self.assertEqual([0.1, -0.6, -0.4, -0.7], sound3.tolist())

        sound1 = self.generator.from_list([0.1, -0.1])
        sound2 = self.generator.from_list([0, 0.5, 0.5, 0.6])
        sound3 = sound2 - sound1
        self.assertEqual([-0.1, 0.6, 0.4, 0.7], sound3.tolist())


    def test_mul(self):

        sound1 = self.generator.from_list([1, -1, 0.5])
        sound2 = self.generator.from_list([0.5, 0.2, 0.5])
        sound3 = sound1 * sound2
        self.assertEqual([0.5, -0.2, 0.25], sound3.tolist())

        sound1 = self.generator.from_list([0.1])
        sound2 = self.generator.from_list([0, 0.5, -1])
        sound3 = sound1 * sound2
        self.assertEqual([0, 0.05, -0.1], sound3.tolist())
        sound4 = sound2 * sound1
        self.assertEqual(sound3, sound4)

        sound1 = self.generator.from_list([0.1, -0.1])
        sound2 = self.generator.from_list([0, 0.5, 0.5, 1])
        sound3 = sound1 * sound2
        self.assertEqual([0, -0.05, 0.05, -0.1], sound3.tolist())
        sound4 = sound2 * sound1
        self.assertEqual(sound3, sound4)


    def test_div(self):

        sound1 = self.generator.from_list([0.5, 0.1, 0.8])
        sound2 = self.generator.from_list([0.5, -0.2, 1])
        sound3 = sound1 / sound2
        self.assertEqual([1, -0.5, 0.8], sound3.tolist())

        sound1 = self.generator.from_list([0.1])
        sound2 = self.generator.from_list([0.5, -0.2, 0.25])
        sound3 = sound1 / sound2
        self.assertEqual([0.2, -0.5, 0.4], sound3.tolist())

        sound1 = self.generator.from_list([0.5])
        sound2 = self.generator.from_list([0, -0.2, 0.5])
        sound3 = sound2 / sound1
        self.assertEqual([0, -0.4, 1], sound3.tolist())


    def test_mix(self):

        sound1 = self.generator.from_list([0.5, 0.6, 0])
        sound2 = self.generator.from_list([0.5, 0.2, 1])
        sound3 = sound1 & sound2
        self.assertEqual([0.5, 0.4, 0.5], sound3.tolist())

        sound1 = self.generator.from_list([0.2])
        sound2 = self.generator.from_list([0.5, -0.2, 0.6])
        sound3 = sound1 & sound2
        self.assertEqual([0.35, 0, 0.4], sound3.tolist())
        sound4 = sound2 & sound1
        self.assertEqual(sound3, sound4)


    def test_concat(self):
        sound1 = self.generator.from_list([0.1])
        sound2 = self.generator.from_list([0.2, 0.3])

        self.assertEqual([0.1, 0.2, 0.3], (sound1 // sound2).tolist())
```

```python


    def test_loop(self):
        sound1 = self.generator.from_list([1])
        sound2 = self.generator.from_list([1, 1, 1])
        self.assertEqual(sound2, sound1.loop(3))

        sound1 = self.generator.from_list([1, 0.5])
        sound2 = self.generator.from_list([1, 0.5, 1, 0.5, 1, 0.5])
        self.assertEqual(sound2, sound1.loop(3))

        self.assertRaises(Exception, lambda : sound1.loop(0))
        self.assertRaises(Exception, lambda : sound1.loop(-1))

    def test_resize(self):
        sound1 = self.generator.from_list([0, 0.1, 0.2])

        self.assertEqual(self.generator.from_list([0]), sound1.resize(1))
        self.assertEqual(self.generator.from_list([0, 0.1]), sound1.resize(2))
        self.assertEqual(self.generator.from_list([0, 0.1, 0.2]), sound1.resize(3))

        self.assertEqual(
            self.generator.from_list([0, 0.1, 0.2, 0.0]),
            sound1.resize(4))

        self.assertEqual(
            self.generator.from_list([0, 0.1, 0.2, 0.0, 0.1]),
            sound1.resize(5))

        self.assertEqual(
            self.generator.from_list([0, 0.1, 0.2, 0.0, 0.1, 0.2]),
            sound1.resize(6))

        self.assertRaises(Exception, lambda : sound1.resize(0))
        self.assertRaises(Exception, lambda : sound1.resize(-1))
        self.assertRaises(Exception, lambda : sound1.resize(0.5))
        self.assertRaises(Exception, lambda : sound1.resize(1.5))


    def test_resample(self):
        sound1 = self.generator.from_list([0, 0.1, 0.2])

        self.assertRaises(Exception, lambda : sound1.resample(0))
        self.assertRaises(Exception, lambda : sound1.resample(-1))
        self.assertRaises(Exception, lambda : sound1.resample(0.5))
        self.assertRaises(Exception, lambda : sound1.resample(1.5))


    def test_copy(self):
        sound1 = self.generator.from_list([0, 0.1, 0.2])
        copy = sound1.copy()

        self.assertEqual(sound1, copy)
        self.assertFalse(sound1 is copy)


    def test_concat(self):
        sound1 = self.generator.from_list([0.1])
        sound2 = self.generator.from_list([0.2])

        self.assertEqual(self.generator.from_list([0.1, 0.2]), sound1.concat(sound2))


    def test_fill(self):
        sound1 = self.generator.from_list([0, 0.1, 0.2])

        self.assertEqual(
            self.generator.from_list([0, 0.1, 0.2] + [0] * (self.beat - 3)),
            sound1.fill(1))
        self.assertEqual(
            self.generator.from_list([0, 0.1, 0.2] + [0] * (self.beat * 2- 3)),
            sound1.fill(2))

        sound2 = self.generator.from_list([0.1] * self.beat)
        self.assertEqual(
            sound2,
            sound2.fill(1))

        self.assertRaises(Exception, lambda : sound1.fill(0))
        self.assertRaises(Exception, lambda : sound1.fill(-1))


    def test_reduce(self):
        sound1 = self.generator.from_list([0, 0.1] * (self.beat - 1))
        self.assertEqual(self.beat, len(sound1.reduce(1)))

        sound1 = self.generator.from_list([0.1] * (self.beat - 1))
        self.assertEqual(len(sound1), len(sound1.reduce(1)))

        sound1 = self.generator.from_list([0, 0.1] * (self.beat - 1))
        self.assertEqual(len(sound1), len(sound1.reduce(2)))

        self.assertRaises(Exception, lambda : sound1.reduce(0))
        self.assertRaises(Exception, lambda : sound1.reduce(-1))


    def test_expand(self):
        sound1 = self.generator.from_list([0, 0.1, 0.2])
        self.assertEqual(self.beat, len(sound1.expand(1)))
        self.assertEqual(self.beat * 2, len(sound1.expand(2)))

        sound1 = self.generator.from_list([0, 0.1] * self.beat)
        self.assertEqual(len(sound1), len(sound1.expand(1)))

        self.assertRaises(Exception, lambda : sound1.expand(0))
        self.assertRaises(Exception, lambda : sound1.expand(-1))


    def test_tolist(self):
        sound1 = self.generator.from_list([0, 0.1, 0.2])
        self.assertEqual([0, 0.1, 0.2], sound1.tolist())


    def test_resample(self):
        sound1 = self.generator.from_list([0, 0.1, 0.2, 0.3])

        self.assertEqual(
            self.generator.from_list([0, 0.1, 0.2, 0.3]),
            sound1.resample(4))

        self.assertEqual(1, len(sound1.resample(1)))
        self.assertEqual(3, len(sound1.resample(3)))
        self.assertEqual(6, len(sound1.resample(6)))
        self.assertEqual(10, len(sound1.resample(10)))

        self.assertRaises(Exception, lambda : sound1.resample(0))
        self.assertRaises(Exception, lambda : sound1.resample(-1))
        self.assertRaises(Exception, lambda : sound1.resample(0.5))
        self.assertRaises(Exception, lambda : sound1.resample(1.5))

class TestCustomCases(MixerTestCase):
    pass

if __name__ == '__main__':
    unittest.main()
```

6. test_parser.py

```python
import minicollider.parser as parser
import unittest

class ParserTestCase(unittest.TestCase):


    def setUp(self):

        self.beat = 8
        self.sample_rate = self.beat * 12

        parser.init(self.sample_rate, self.beat, 0)
        self.generator = parser.generator


    def assertParseEqualList(self, list, input):
        self.assertParseEqual(self.generator.from_list(list), input)


    def assertParseFail(self, input):
        self.assertRaises(Exception, lambda : parser.parse(input),
            'Se esperaba un error para el input: %s' % input)


    def assertParseAllFail(self, input_list):
        for input in input_list:
            self.assertParseFail(input)


    def assertParseEqual(self, sound, input):
        parsed_sound = parser.parse(input)
        self.assertEqual(sound, parsed_sound, '%s != %s' % (sound, parsed_sound))


    def assertParseAllEqual(self, sound, input_list):
        for input in input_list:
            self.assertParseEqual(sound, input)


    def assertElementsInRange(self, list, min, max):
        for item in list:
            self.assertTrue(min <= item <= max)


class TestGeneratorsCases(ParserTestCase):


    def test_manual(self):
        self.assertParseAllEqual(
            self.generator.from_list([0]),
            ['0', '{0}', '{ 0}', '{0 }', '{ 0 }']
        )

        self.assertParseAllEqual(
            self.generator.from_list([1]),
            ['1', '{1}', '{ 1}', '{1 }', '{ 1 }']
        )

        self.assertParseAllEqual(
            self.generator.from_list([-1]),
            ['-1', '{-1}', '{ -1}', '{-1 }', '{ -1 }']
        )

        self.assertParseAllEqual(
            self.generator.from_list([1]),
            ['1.0', '{1.0}', '{ 1.0}', '{1.0 }', '{ 1.0 }']
        )

        self.assertParseAllEqual(
            self.generator.from_list([-0.5]),
            ['-0.5', '{-0.5}', '{ -0.5}', '{-0.5 }', '{ -0.5 }']
        )

        self.assertParseAllFail(
            ['{', '}', '{ }', '}{', 'a', '-a', 'asd1']
        )

        self.assertParseAllFail(
            ['2', '2.0', '1.1', '-2', '-2.1', '0.1.0']
        )

        self.assertParseAllFail(
            ['{2}', '{2.0}', '{1.1}', '{-2}', '{-2.1}']
        )


    def test_silence(self):
        self.assertParseAllEqual(self.generator.silence(),
            ['sil', 'sil()', '{sil}', '{sil()}']
        )

        self.assertParseAllEqual(self.generator.silence(),
            ['silence', 'silence()', '{silence}', '{silence()}']
        )

        self.assertParseAllFail(['Sil', 'sile()', 'Silence', 'sil ence'])

        self.assertParseAllFail(['sil(1)', 'sil(1,2)'])


    def test_sine(self):
        self.assertParseEqual(self.generator.sine(1, 1), 'sin(1)')
        self.assertParseEqual(self.generator.sine(5, 1), 'sin(5)')
        self.assertParseEqual(self.generator.sine(11, 1), 'sin(11)')

        self.assertParseEqual(self.generator.sine(1, 0.5), 'sin(1, 0.5)')
        self.assertParseEqual(self.generator.sine(1, 0), 'sin(1, 0)')
        self.assertParseEqual(self.generator.sine(10, 0.2), 'sin(10, 0.2)')

        self.assertParseAllFail(['sin', 'sin()', 'sin(0)','sin(1.0)', 'sin(-1)'])
        self.assertParseAllFail(['sin(1, 0)', 'sin(1, 2)', 'sin(1, -1)'])


    def test_linear(self):
        self.assertParseAllEqual(self.generator.linear(0, 1),
            ['linear(0, 1)', 'lin(0, 1)'])

        self.assertParseAllEqual(self.generator.linear(0.5, 0.5),
            ['linear(0.5, 0.5)', 'lin(0.5, 0.5)'])

        self.assertParseAllEqual(self.generator.linear(-0.5, 0.5),
            ['linear(-0.5, 0.5)', 'lin(-0.5, 0.5)'])

        self.assertParseAllEqual(self.generator.linear(0.5, -0.5),
            ['linear(0.5, -0.5)', 'lin(0.5, -0.5)'])

        self.assertParseAllEqual(self.generator.linear(-0.5, -0.5),
            ['linear(-0.5, -0.5)', 'lin(-0.5, -0.5)'])

        self.assertParseAllFail(
            ['linear', 'linear()', 'lin', 'lin()', 'lin(0)', 'lin(0, 0, 0)'])

        self.assertParseAllFail(
            ['lin(2, 0)', 'lin(0, 2)', 'lin(-2, 0)', 'lin(0, -2)',])


    def test_noise(self):
        sound = parser.parse('noi')
        self.assertEqual(self.beat, len(sound))

        sound = parser.parse('noi()')
        self.assertEqual(self.beat, len(sound))

        sound = parser.parse('noise')
        self.assertEqual(self.beat, len(sound))

        sound = parser.parse('noise()')
        self.assertEqual(self.beat, len(sound))

        sound = parser.parse('noi(0)')
        self.assertEqual(self.generator.silence(), sound)

        sound = parser.parse('noi(0.5)')
        self.assertEqual(self.beat, len(sound))
        self.assertElementsInRange(sound, -0.5, 0.5)

        self.assertParseAllFail(['noi(2)', 'noi(-2)'])


class TestOperatorCases(ParserTestCase):


    def test_add(self):

        self.assertParseEqualList([3], '1 + 2')
        self.assertParseEqualList([6], '1 + 2 + 3')
        self.assertParseEqualList([-1], '1 + 2 + -4')
        self.assertParseEqualList([2, 5, -1], '{1; 2; 3} + {1; 3; -4}')
        self.assertParseEqualList([2, 5, -1], '{1; 4; -2} + {1}')
        self.assertParseEqualList([2, 5, -1], '{1; 4; -2} + 1')
        self.assertParseEqualList([2, 5, -1, 0], '{1; 3; -2; -2} + {1; 2}')
        self.assertParseEqualList([2, 5, -1, 0], '{1; 2} + {1; 3; -2; -2}')


    def test_sub(self):

        self.assertParseEqualList([-1], '1 - 2')
        self.assertParseEqualList([-4], '1 - 2 - 3')
        self.assertParseEqualList([3], '1 - 2 - -4')
        self.assertParseEqualList([0, -1, 7], '{1; 2; 3} - {1; 3; -4}')
        self.assertParseEqualList([0, 3, -3], '{1; 4; -2} - {1}')
        self.assertParseEqualList([0, 3, -3], '{1; 4; -2} - 1')
        self.assertParseEqualList([0, 1, -3, -4], '{1; 3; -2; -2} - {1; 2}')
        self.assertParseEqualList([0, -1, 3, 4], '{1; 2} - {1; 3; -2; -2}')


    def test_mul(self):

        self.assertParseEqualList([2], '1 * 2')
        self.assertParseEqualList([6], '1 * 2 * 3')
        self.assertParseEqualList([-8], '1 * 2 * -4')
        self.assertParseEqualList([1, 6, -12], '{1; 2; 3} * {1; 3; -4}')
        self.assertParseEqualList([2, 8, -4], '{1; 4; -2} * {2}')
        self.assertParseEqualList([2, 8, -4], '{1; 4; -2} * 2')
        self.assertParseEqualList([1, 6, -2, -4], '{1; 3; -2; -2} * {1; 2}')
        self.assertParseEqualList([1, 6, -2, -4], '{1; 2} * {1; 3; -2; -2}')


    def test_div(self):

        self.assertParseEqualList([5], '10 / 2')
        self.assertParseEqualList([2], '12 / 2 / 3')
        self.assertParseEqualList([-2], '16 / 2 / -4')
        self.assertParseEqualList([3, 0.5, -0.25], '{3; 2; 1} / {1; 4; -4}')
        self.assertParseEqualList([5, 2, -0.5], '{10; 4; -1} / {2}')
        self.assertParseEqualList([5, 2, -0.5], '{10; 4; -1} / 2')
        self.assertParseEqualList([1, 1.5, -2, -1], '{1; 3; -2; -2} / {1; 2}')
        self.assertParseEqualList([1, 1, -0.5, -1.5], '{1; 3} / {1; 3; -2; -2}')


    def test_div(self):

        self.assertParseEqualList([5], '10 / 2')
        self.assertParseEqualList([2], '12 / 2 / 3')
        self.assertParseEqualList([-2], '16 / 2 / -4')
        self.assertParseEqualList([3, 0.5, -0.25], '{3; 2; 1} / {1; 4; -4}')
        self.assertParseEqualList([5, 2, -0.5], '{10; 4; -1} / {2}')
        self.assertParseEqualList([5, 2, -0.5], '{10; 4; -1} / 2')
        self.assertParseEqualList([1, 1.5, -2, -1], '{1; 3; -2; -2} / {1; 2}')
        self.assertParseEqualList([1, 1, -0.5, -1.5], '{1; 3} / {1; 3; -2; -2}')


    def test_mix(self):

        self.assertParseEqualList([6], '10 & 2')
        self.assertParseEqualList([5], '12 & 2 & 3')
        self.assertParseEqualList([2.5], '16 & 2 & -4')
        self.assertParseEqualList([2, 3, -1.5], '{3; 2; 1} & {1; 4; -4}')
        self.assertParseEqualList([6, 3, 0.5], '{10; 4; -1} & {2}')
        self.assertParseEqualList([6, 3, 0.5], '{10; 4; -1} & 2')
        self.assertParseEqualList([1, 2.5, -0.5, 0], '{1; 3; -2; -2} & {1; 2}')
        self.assertParseEqualList([1, 2.5, -0.5, 0], '{1; 2} & {1; 3; -2; -2}')


    def test_concat(self):

        self.assertParseEqualList([10, 2], '10 ; 2')
        self.assertParseEqualList([10, 2, -3], '10 ; 2 ; -3')

        self.assertParseEqualList([10, 2], '{10} ; {2}')
        self.assertParseEqualList([10, 1, 2], '{10; 1} ; {2}')
        self.assertParseEqualList([10, 2, 1], '{10} ; {2; 1}')
        self.assertParseEqualList([1, 2, 3, 4, 5, 6], '{1; 2} ; {3; 4} ; {5; 6}')


class TestMethodsCases(ParserTestCase):


    def test_loop(self):

        self.assertParseEqualList([5], '5.loop()')
        self.assertParseEqualList([5], '5.loop')

        self.assertParseEqualList([1, 1], '1.loop(2)')
        self.assertParseEqualList([1, 1], '{1}.loop(2)')

        self.assertParseEqualList([1, 2], '{1 ; 2}.loop(1)')
        self.assertParseEqualList([1, 2] * 2, '{1 ; 2}.loop(2)')
        self.assertParseEqualList([1, 2] * 10, '{1 ; 2}.loop(10)')

        self.assertParseEqualList([1], '{1 ; 2}.loop(0.5)')
        self.assertParseEqualList([1, 2, 1], '{1 ; 2}.loop(1.5)')

        self.assertParseFail('1.loop(0)')
        self.assertParseFail('1.loop(-1)')


    def test_fill(self):

        self.assertParseEqualList([1] + [0] * (self.beat - 1), '1.fill(1)')
        self.assertParseEqualList([1] + [0] * (self.beat - 1), '1.fill()')
        self.assertParseEqualList([1] + [0] * (self.beat - 1), '1.fill')

        self.assertParseEqualList([1, 2, 3] + [0] * (self.beat - 3), '{1;2;3}.fill')
        self.assertParseEqualList([1, 2, 3] + [0] * (self.beat * 2 - 3), '{1;2;3}.fill(2)')

        self.assertParseEqualList([1 ] + [0] * (self.beat / 2 - 1), '1.fill(0.5)')
        self.assertParseEqualList([1, 2, 3] * 1000, '{1;2;3}.loop(1000).fill')

        self.assertParseFail('1.fill(0)')
        self.assertParseFail('1.fill(-1)')


    def test_reduce(self):

        self.assertParseEqualList([1], '1.reduce')
        self.assertParseEqualList([1], '1.reduce()')
        self.assertParseEqualList([1, 2] * (self.beat / 2), '{1; 2}.loop(100).reduce')
        self.assertParseEqualList([2] * self.beat, '2.loop(1000).reduce()')
        self.assertParseEqualList([2] * (self.beat * 2), '2.loop(1000).reduce(2)')

        self.assertParseFail('1.reduce(0)')
        self.assertParseFail('1.reduce(-1)')


    def test_expand(self):

        self.assertParseEqualList([1] * self.beat, '1.expand')
        self.assertParseEqualList([1] * self.beat, '1.expand()')
        self.assertParseEqualList([1] * (self.beat / 2) + [2] * (self.beat / 2), '{1;2}.expand()')
        self.assertParseEqualList([1] * self.beat + [2] * self.beat , '{1;2}.expand(2)')
        self.assertParseEqualList([1] * 1000, '1.loop(1000).expand()')

        self.assertParseFail('1.expand(0)')
        self.assertParseFail('1.expand(-1)')


class TestPrecedenceCases(ParserTestCase):


    def test_add_mul(self):
        self.assertParseEqualList([7], '1 + 2 * 3')
        self.assertParseEqualList([7], '2 * 3 + 1')
        self.assertParseEqualList([8], '2 * {3 + 1}')
        self.assertParseEqualList([8], '{3 + 1} * 2')


    def test_add_div(self):

        self.assertParseEqualList([4], '1 + 9 / 3')
        self.assertParseEqualList([3], '6 / 3 + 1')
        self.assertParseEqualList([2], '8 / {3 + 1}')
        self.assertParseEqualList([2], '{3 + 1} / 2')


    def test_add_sub(self):

        self.assertParseEqualList([0], '1 + 2 - 3')
        self.assertParseEqualList([2], '1 - 2 + 3')
        self.assertParseEqualList([-4], '1 - 2 + -3')
        self.assertParseEqualList([6], '1 + 2 - -3')


    def test_add_concat(self):

        self.assertParseEqualList([6, 3], '1 + 5 ; 3')
        self.assertParseEqualList([6, 3, 1.5], '1 + 5 ; 3 ; 0.5 + 1')


    def test_sub_concat(self):

        self.assertParseEqualList([-4, 3], '1 - 5 ; 3')
        self.assertParseEqualList([-4, 3, -0.5], '1 - 5 ; 3 ; 0.5 - 1')


    def test_mul_concat(self):

        self.assertParseEqualList([10, 3], '2 * 5 ; 3')
        self.assertParseEqualList([10, 3, 1], '2 * 5 ; 3 ; 0.5 * 2')


    def test_div_concat(self):

        self.assertParseEqualList([2, 3], '10 / 5 ; 3')
        self.assertParseEqualList([2, 3, 0.25], '10 / 5 ; 3 ; 0.5 / 2')


    def test_loop_concat(self):

        self.assertParseEqualList([1, 1, 1, 3], '1.loop(3) ; 3')
        self.assertParseEqualList([3, 1, 1, 1], '3 ; 1.loop(3)')


    def test_expand_concat(self):

        self.assertParseEqualList([1] * self.beat + [3], '1.expand ; 3')
        self.assertParseEqualList([3] + [1] * self.beat, '3 ; 1.expand')


    def test_reduce_concat(self):

        self.assertParseEqualList([1, 2, 3], '{1;2}.reduce ; 3')
        self.assertParseEqualList([3, 1, 2], '3 ; {1;2}.reduce')


    def test_fill_concat(self):

        self.assertParseEqualList([1, 2] + [0] * (self.beat - 2) + [3], '{1;2}.fill ; 3')
        self.assertParseEqualList([3] + [1, 2] + [0] * (self.beat - 2), '3 ; {1;2}.fill')


    def test_add_mix(self):

        self.assertParseEqualList([4.5], '1 + 5 & 3')
        self.assertParseEqualList([4], '3 & 4 + 1')

    def test_loop_mix(self):

        self.assertParseEqualList([2, 2, 2], '1.loop(3) & 3')
        self.assertParseEqualList([2, 2, 2], '3 & 1.loop(3)')


class TestCustomCases(ParserTestCase):


    def test_enunciado(self):

        self.assertParseEqualList([1, 2], '1; 2')
        self.assertParseEqualList([1, 2], '{1; 2}')
        self.assertParseEqualList([7], '{1 + 2*3}')
        self.assertParseEqualList([6, 3, 6], '{{2;1} mul {3;3;3}}')
        self.assertParseEqualList([0, 1], '{1-1;2-1}')
        self.assertParseEqualList([2, 3, 4], '{4+1*-2 & {2;4;6}}')
        self.assertParseEqualList([0, 1, 1], '{0;1.loop(2)}')
        self.assertParseEqualList([0, 1], '{0;1;2;3}.loop(0.5)')


if __name__ == '__main__':
    unittest.main()
```