DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Grupo Fanstama

8 de Julio de 2015 — Teoría de Lenguajes

Integrante	LU	Correo electrónico
Vallejo, Nicolás Agustín	500/10	nicopr08@gmail.com
Tastzian, Juan Manuel	39/10	jm@tast.com.ar
González, Pablo Gabriel	476/10	pablo.gonzalez.alba@gmail.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

1. Introducción

El presente trabajo práctico consiste en implementar diversos elementos vistos en la materia para, en última instancia, poder traducir un lenguaje de composición musical llamado **Musileng** que tendrá como salida un archivo que se transformará a formato **MIDI** que se puede escuchar en cualquier reproductor que soporte dicho estándar.

Dichos elementos implementados por el grupo, son:

- **Gramática**: Primero se creó la gramática que representará la forma de generar cualquier cadena aceptada por nuestro lenguaje.
- **Lexer**: En base a la gramática, se hizo el Lexer. El Lexer se encarga de separar la cadena de entrada en distintos **tokens**.
- **Parser**: La salida del Lexer es utilizada por el **Parser**, que se encarga de generar el **AST** (*Abstract Syntax Tree*).
- **Traductor**: Finalmente mediante el **Traductor**, se genera el archivo de salida necesario para poder utilizarlo como entrada del programa generador de MIDI llamado **midicomp**.

En las secciones subsiguientes se explica con más detalle cada una de estas partes desarrolladas en el trabajo práctico.

2. Solución del Problema

Para crear el traductor se dividió la solución en cuatro partes: una Gramática, un Lexer, un Parser y un Traductor. Se programó con **Python** utilizando la librería **Ply**.

2.1. Lexer

El Lexer traduce la cadena original a una cadena tokenizada asignando valores a cada token.

- Los siguientes tokens se traducen sin cambios: '(', ')', '{', '}', ';', ',', '=', '/', '.', 'voz', 'repetir', 'compas', '#tempo', '#compas', 'const', 'nota', 'silencio'.
- Los números se traducen al token $NUMBER$.
- Las figuras: 'redonda', 'blanca', 'negra', 'corchea', 'semicorchea', 'fusa' y 'semifusa' se traducen al token $FIGURE$.
- Las notas: 'do', 're', 'mi', 'fa', 'sol', 'la' y 'si' se traducen al token $NOTE$.
- Los símbolos '+' y '-' se traducen al token $NOTE\_MODIFIER$
- Los comentarios iniciando con '//' hasta el fin de línea se eliminan.
- Y cualquier otra cadena que comience con una letra y contenga únicamente letras, números y '_' se traduce a $CONST\_NAME$

A continuación se muestra el código de *lexer_rules.py* que define estos tokens para **Ply**:

```
# -*- coding: utf-8 -*-
tokens = [
    'TEMPO_DEFINITION',
    'BAR_DEFINITION',
    'CONST_DEFINITION',
    'FIGURE',
    'NUMBER',
    'SLASH',
    'EQUAL',
    'SEMICOLON',
    'LPAREN',
    'RPAREN',
    'LBRACE',
    'RBRACE',
    'VOICE_BLOCK',
    'REPEAT_BLOCK',
    'BAR_BLOCK',
    'NOTE_CALL',
    'SILENCE',
    'DOT',
    'NOTE',
    'NOTE_MODIFIER',
    'CONST_NAME',
    'COMMA'
]

t_LPAREN = r"\("
t_RPAREN = r"\)"
t_LBRACE = r"\{"
t_RBRACE = r"\}"
t_SEMICOLON = r"\;"
t_EQUAL = r"\="
t_SLASH = r"\/"
t_VOICE_BLOCK = r"voz"
```

```
t_REPEAT_BLOCK = r"repetir"
t_BAR_BLOCK = r"compas"
t_TEMPO_DEFINITION = r"\#tempo"
t_BAR_DEFINITION = r"\#compas"
t_CONST_DEFINITION = r"const"
t_NOTE_CALL = r"nota"
t_SILENCE = r"silencio"
t_DOT = r"\."
t_NOTE_MODIFIER = r"(\+|\-)"
# Cualquier nombre excepto las keywords de la gramática
t_CONST_NAME = r"(?!(const|voz|nota|repetir|compas|silencio))[a-zA-Z][_a-zA-Z0-9]*"
t_COMMA = r","

t_ignore = " \t"

def t_NUMBER(token):
    r"[0-9]+"
    token.value = int(token.value)
    return token

def t_FIGURE(token):
    r"(redonda|blanca|negra|corchea|semicorchea|fusa|semifusa)"
    return token

def t_NOTE(token):
    r"(?!(redonda|repetir|silencio))(do|re|mi|fa|sol|la|si)"
    # La negación es necesaria porque sino toma las notas primero
    # Ya se probó cambiar el orden y no funcionó
    return token

def t_NEWLINE(token):
    r"\n+"
    token.lexer.lineno += len(token.value)

def t_IGNORE_COMMENTS(token):
    r"//(.*)\n"
    token.lexer.lineno += 1

def t_error(token):
    raise Exception("Error de sintaxis: Token desconocido en línea {0}. \"{1}\"".format(
        token.lineno, token.value.partition("\n")[0]))
```

2.2. Parser

El Parser toma la cadena ya tokenizada y genera el **Abstract Syntax Tree**.

Las producciones especificadas son las de la Gramática, cada función equivale a una producción (sin utilizar pipes '|').

A continuación se muestra el código de *parser_rules.py* que define estas producciones para **Ply**:

```
# -*- coding: utf-8 -*-
from lexer_rules import tokens

from expressions import *

constants = {}

def p_start(subexpressions):
    'start : tempo_definition bar_definition constants voices'
    subexpressions[0] = Start(subexpressions[1], subexpressions[2], subexpressions[4])

def p_tempo_definition(subexpressions):
    'tempo_definition : TEMPO_DEFINITION FIGURE NUMBER'
    subexpressions[0] = TempoDefinition(subexpressions[2], subexpressions[3], subexpressions.lineno(1))

def p_bar_definition(subexpressions):
    'bar_definition : BAR_DEFINITION NUMBER SLASH NUMBER'
    subexpressions[0] = BarDefinition(subexpressions[2], subexpressions[4], subexpressions.lineno(1))

def p_constants_empty(subexpressions):
    'constants :'
    pass

def p_constants(subexpressions):
    'constants : CONST_DEFINITION CONST_NAME EQUAL NUMBER SEMICOLON constants'
    name = subexpressions[2]
    value = subexpressions[4]

    if name in constants:
        raise Exception("Constante redefinida: {0}. Primera vez definida en línea {1}".format(
            name, subexpressions.lineno(1)))
    constants[name] = value

# Las siguientes producciones:
#     Voices -> Voice MaybeVoices
#     MaybeVoices -> lambda | Voice MaybeVoices
# Son para requerir que exista 1 voz al nivel de la gramática

def p_voices(subexpressions):
    'voices : voice maybe_voices'
    subexpressions[0] = Voices(subexpressions[1], subexpressions[2])

def p_maybe_voices_empty(subexpressions):
    'maybe_voices :'
    pass

def p_maybe_voices(subexpressions):
    'maybe_voices : voice maybe_voices'
    subexpressions[0] = Voices(subexpressions[1], subexpressions[2])

def p_voice_number(subexpressions):
    'voice : VOICE_BLOCK LPAREN NUMBER RPAREN LBRACE bars RBRACE'
```

```
    subexpressions[0] = Voice(subexpressions[3], subexpressions[6], subexpressions.lineno(1))

def p_voice_constant(subexpressions):
    'voice : VOICE_BLOCK LPAREN CONST_NAME RPAREN LBRACE bars RBRACE'
    if subexpressions[3] not in constants:
        raise Exception("Constante no definida: {0}. Utilizada en línea {1}".format(
            subexpressions[3], subexpressions.lineno(1)))
    subexpressions[0] = Voice(constants[subexpressions[3]], subexpressions[6], subexpressions.lineno(1))

# Con Bars se hace lo mismo que con Voices para requerir al menos un compas

def p_bars(subexpressions):
    'bars : bar maybe_bars'
    subexpressions[0] = Bars(subexpressions[1], subexpressions[2])

def p_maybe_bars_empty(subexpressions):
    'maybe_bars :'
    pass

def p_maybe_bars(subexpressions):
    'maybe_bars : bar maybe_bars'
    subexpressions[0] = Bars(subexpressions[1], subexpressions[2])

def p_bar_repeat(subexpressions):
    'bar : repeat'
    subexpressions[0] = subexpressions[1]

def p_bar(subexpressions):
    'bar : BAR_BLOCK LBRACE notes RBRACE'
    subexpressions[0] = Bar(subexpressions[3], subexpressions.lineno(1))

def p_repeat_number(subexpressions):
    'repeat : REPEAT_BLOCK LPAREN NUMBER RPAREN LBRACE bars RBRACE'
    subexpressions[0] = Repeat(subexpressions[3], subexpressions[6], subexpressions.lineno(1))

def p_repeat_constant(subexpressions):
    'repeat : REPEAT_BLOCK LPAREN CONST_NAME RPAREN LBRACE bars RBRACE'
    if subexpressions[3] not in constants:
        raise Exception("Constante no definida: {0}. Utilizada en línea {1}".format(
            subexpressions[3], subexpressions.lineno(1)))
    subexpressions[0] = Repeat(constants[subexpressions[3]], subexpressions[6], subexpressions.lineno(1))

def p_notes_empty(subexpressions):
    'notes :'
    pass

def p_notes_silence(subexpressions):
    'notes : SILENCE LPAREN FIGURE maybe_dot RPAREN SEMICOLON notes'
    subexpressions[0] = Silence(subexpressions[3], subexpressions[4], subexpressions[7])

def p_notes_note_number(subexpressions):
    'notes : NOTE_CALL LPAREN NOTE maybe_note_modifier COMMA NUMBER COMMA FIGURE maybe_dot RPAREN
        SEMICOLON notes'
    subexpressions[0] = Note(subexpressions[3], subexpressions[4], subexpressions[6], subexpressions[8],
        subexpressions[9], subexpressions[12], subexpressions.lineno(1))

def p_notes_note_constant(subexpressions):
    'notes : NOTE_CALL LPAREN NOTE maybe_note_modifier COMMA CONST_NAME COMMA FIGURE maybe_dot RPAREN
```

```
        SEMICOLON notes'
    if subexpressions[6] not in constants:
        raise Exception("Constante no definida: {0}. Utilizada en línea {1}".format(
            subexpressions[6], subexpressions.lineno(1)))
    subexpressions[0] = Note(subexpressions[3], subexpressions[4], constants[subexpressions[6]],
        subexpressions[8], subexpressions[9], subexpressions[12], subexpressions.lineno(1))

def p_maybe_dot_empty(subexpressions):
    'maybe_dot :'
    pass

def p_maybe_dot(subexpressions):
    'maybe_dot : DOT'
    subexpressions[0] = Dot(subexpressions[1])

def p_maybe_note_modifier_empty(subexpressions):
    'maybe_note_modifier :'
    pass

def p_maybe_note_modifier(subexpressions):
    'maybe_note_modifier : NOTE_MODIFIER'
    subexpressions[0] = NoteModifier(subexpressions[1])

def p_error(subexpressions):
    raise Exception("Error de sintaxis en línea {0}".format(subexpressions.lineno))
```

2.3. Traductor

El traductor recibe el **Abstract Syntax Tree** que, en este caso, equivale a una instancia de la clase **Start** y lo recorre para ir traduciendo al lenguaje intermedio para que **midicomp** lo pueda traducir a $MIDI$.

Se decidió que dónde se espera un bloque compás (clase *Bar*) pueda ir un bloque de repetir (clase *Repeat*), permitiendo repetir anidadamente con el resultado esperado.

La clase *Repeat* entonces responde al método *get_arr_bars*() para devolver los compases hijos, repetiéndolos las veces correspondientes.

También se decidió que dónde se espera una nota (clase *Note*), pueda ir un silencio (clase *Silence*).

Luego se chequeará si es una nota para escribir en el track, pero ambos devuelven su valor para aumentar el tiempo transcurrido.

A continuación se muestra el código de *expressions.py* que contiene las clases que conforman el **AST**:

```
# -*- coding: utf-8 -*-
figure_values = {
    "redonda": 1,
    "blanca": 2,
    "negra": 4,
    "corchea": 8,
    "semicorchea": 16,
    "fusa": 32,
    "semifusa": 64
}

class Start(object):
    def __init__(self, tempo, bar, voices):
        self.tempo = tempo
        self.bar = bar
        self.voices = voices

        arr_voices = self.get_voices()
        if len(arr_voices) > 16:
            raise Exception(
                "No se permiten más de 16 voces. Línea {0}".format(arr_voices[16].line))

        for voice in arr_voices:
            for bar in voice.get_bars():
                if bar.get_value() < self.bar.get_value():
                    raise Exception(
                        "Compás con duración ({0}) más corta que la indicada ({1}). Línea {2}".format(
                            bar.get_value(), self.bar.get_value(), bar.line))
                if bar.get_value() > self.bar.get_value():
                    raise Exception(
                        "Compás con duración ({0}) más larga que la indicada ({1}). Línea {2}".format(
                            bar.get_value(), self.bar.get_value(), bar.line))

    def name(self):
        return "start"

    def children(self):
        return [self.tempo, self.bar, self.voices]

    def get_voices(self):
        return self.voices.get_arr_voices();

class TempoDefinition(object):
    def __init__(self, figure, speed, line):
        self.figure = figure
        self.speed = speed
```

```
        if speed == 0:
            raise Exception(
                "Cantidad de repeticiones por minuto incorrecta. Debe ser mayor a 0. Línea {0}".format(
                    line))

    def name(self):
        return "#tempo " + self.figure + " " + str(self.speed)

    def milliseconds(self):
        return int(1000000 * 60 * figure_values[self.figure] / (4 * float(self.speed)))

    def children(self):
        return []

class BarDefinition(object):
    def __init__(self, beat, figure, line):
        self.beat = beat
        self.figure = figure

        if beat == 0:
            raise Exception("Cantidad de pulsos incorrecta. Debe ser mayor a 0. Línea {0}".format(line))

        if figure not in figure_values.values():
            raise Exception(
                "Pulso incorrecto. Debe ser el valor de una figura: 1, 2, 4, 8, 16, 32, 64. Línea {0}"
                    .format(line))

    def name(self):
        return "#compas " + self.fraction()

    def fraction(self):
        return str(self.beat) + "/" + str(self.figure)

    def children(self):
        return []

    def get_value(self):
        return self.beat * (1 / float(self.figure))

class Voices(object):
    def __init__(self, voice, other_voices):
        self.voice = voice
        self.other_voices = other_voices

    def name(self):
        return "voces"

    def children(self):
        if self.other_voices == None:
            return [self.voice]
        else:
            return [self.voice, self.other_voices]

    def get_arr_voices(self):
        voices = [self.voice]
        if self.other_voices is not None:
            voices += self.other_voices.get_arr_voices()
```

```
        return voices

class Voice(object):
    def __init__(self, voice_number, bars, line):
        self.voice_number = voice_number
        self.bars = bars
        self.line = line

        if voice_number >= 128:
            raise Exception("Instrumento inválido. Debe estar entre 0 y 127. Línea {0}".format(line))

    def name(self):
        return "voz " + str(self.voice_number)

    def children(self):
        return [self.bars]

    def get_bars(self):
        return self.bars.get_arr_bars()

class Bars(object):
    def __init__(self, bar, other_bars):
        self.bar = bar
        self.other_bars = other_bars

    def name(self):
        return "compases"

    def children(self):
        if self.other_bars == None:
            return [self.bar]
        else:
            return [self.bar, self.other_bars]

        return voices

    def get_arr_bars(self):
        bars = self.bar.get_arr_bars()
        if self.other_bars is not None:
            bars += self.other_bars.get_arr_bars()

        return bars

class Bar(object):
    def __init__(self, notes, line):
        self.notes = notes
        self.line = line

    def name(self):
        return "compas"

    def children(self):
        return [self.notes]

    def get_arr_bars(self):
        return [self]
```

```
    def get_notes(self):
        return self.notes.get_arr_notes()

    def get_value(self):
        value = 0
        for note in self.get_notes():
            value +=  1 / float(note.get_value())
            if note.dot is not None:
                value += 0.5 / float(note.get_value())

        return value

# Un repetir puede ir en lugar de un compás, por lo tanto también implementa
# get_arr_bars()

class Repeat(object):
    def __init__(self, times, bars, line):
        self.times = times
        self.bars = bars

        if times == 0:
            raise Exception(
                "Número de repeticiones incorrecto. Debe haber al menos 1. Línea {0}".format(line))

    def name(self):
        return "repetir " + str(self.times)

    def children(self):
        return [self.bars]

    def get_arr_bars(self):
        return self.bars.get_arr_bars() * self.times

class Dot(object):
    def __init__(self, dot):
        self.dot = dot

    def name(self):
        return str(self.dot)

    def children(self):
        return []

class NoteModifier(object):
    def __init__(self, note_modifier):
        self.note_modifier = note_modifier

    def name(self):
        return str(self.note_modifier)

    def children(self):
        return []

class Silence(object):
    def __init__(self, figure, dot, other_notes):
        self.figure = figure
        self.dot = dot
        self.other_notes = other_notes
```

```
    def name(self):
        return "silencio " + self.figure

    def children(self):
        result = []
        if self.dot is not None:
            result.append(self.dot)
        if self.other_notes is not None:
            result.append(self.other_notes)
        return result

    def get_arr_notes(self):
        notes = [self]
        if self.other_notes is not None:
            notes += self.other_notes.get_arr_notes()

        return notes

    def get_value(self):
        return figure_values[self.figure]

class Note(object):
    def __init__(self, note, note_modifier, octave, figure, dot, other_notes, line):
        self.note = note
        self.note_modifier = note_modifier
        self.octave = octave
        self.figure = figure
        self.dot = dot
        self.other_notes = other_notes

        if octave not in range(1,10):
            raise Exception("Octava incorrecta. Debe estar entre 1 y 9. Línea {0}".format(line))

    def name(self):
        return "nota " + self.note + self.note_modifier_name() + " octava " + str(self.octave)
            + self.figure

    def children(self):
        result = []
        if self.note_modifier is not None:
            result.append(self.note_modifier)
        if self.dot is not None:
            result.append(self.dot)
        if self.other_notes is not None:
            result.append(self.other_notes)
        return result

    def get_arr_notes(self):
        notes = [self]
        if self.other_notes is not None:
            notes += self.other_notes.get_arr_notes()

        return notes

    def note_modifier_name(self):
        note_modifier = ""
        if self.note_modifier is not None:
```

```
        note_modifier = self.note_modifier.name()

    return note_modifier

def __str__(self):
    translation_notes = {
        "do": "c",
        "re": "d",
        "mi": "e",
        "fa": "f",
        "sol": "g",
        "la": "a",
        "si": "b"
    }

    return translation_notes[self.note] + self.note_modifier_name() + str(self.octave)

def get_value(self):
    return figure_values[self.figure]
```

Y a continuación se muestra parte del código de *musileng* que contiene la traducción del **AST** al lenguaje que entiende **midicomp**:

```
def translate_to_txt_midi(ast, output):
    write_header(ast, output)

    for idx, voice in enumerate(ast.get_voices()):
        write_voice(ast, idx, voice, output)


def write_header(ast, output):
    num_tracks = len(ast.get_voices()) + 1
    output.write("MFile 1 {0} {1}\n".format(num_tracks, clicks_per_beat))
    output.write("MTrk\n")
    output.write("000:00:000 TimeSig {0} 24 8\n".format(ast.bar.fraction()))
    output.write("000:00:000 Tempo {0}\n".format(ast.tempo.milliseconds()))
    output.write("000:00:000 Meta TrkEnd\n")
    output.write("TrkEnd\n")

def write_voice(ast, voice_idx, voice, output):
    output.write("MTrk\n")
    output.write("000:00:000 Meta TrkName \"Voz {0}\"\n".format(voice_idx + 1))
    output.write("000:00:000 ProgCh ch={0} prog={1}\n".format(voice_idx + 1, voice.voice_number))

    for bar_idx, bar in enumerate(voice.get_bars()):
        last_bar, last_beat, last_click = write_bar(ast, voice_idx, bar_idx, bar, output)

    output.write("%03d:%02d:%03d Meta TrkEnd\n" % (last_bar, last_beat, last_click))
    output.write("TrkEnd\n")

def write_bar(ast, voice_idx, bar_idx, bar, output):
    beat = 0
    click = 0
    for note in bar.get_notes():
        # note podría ser una nota o un silencio
        if note.__class__.__name__ is 'Note':
            output.write("%03d:%02d:%03d On     ch=%d note=%s vol=70\n" %
                (bar_idx, beat, click, voice_idx + 1, note))

        # La duración se calcula haciendo regla de 3 del valor contra la figura
        # de la definición del compás
        duration = int(clicks_per_beat * ast.bar.figure / float(note.get_value()))
        # Y se agrega media figura si hay un puntillo
        if note.dot is not None:
            duration += int(clicks_per_beat * ast.bar.figure / float(note.get_value() * 2))
        # Se aumentan los clicks, pulsos y el compás, teniendo en cuenta que
        # los clicks son módulo 384 y los pulsos según lo que se indique en la
        # definición del compás
        click += duration
        beat += click / clicks_per_beat
        click %= clicks_per_beat
        bar_idx += beat / ast.bar.beat
        beat %= ast.bar.beat

        if note.__class__.__name__ is 'Note':
            output.write("%03d:%02d:%03d Off    ch=%d note=%s vol=0\n" %
                (bar_idx, beat, click, voice_idx + 1, note))

    return bar_idx, beat, click
```

3. Gramática

La Gramática desarrolada es la siguiente: $\langle V_n,\ V_t,\ Producciones,\ Start \rangle$

3.1. Símbolos No Terminales

Start, TempoDefinition, BarDefinition, Constants, Voices, MaybeVoices, Voice, Bars, MaybeBars, Bar, Repeat, Notes, MaybeDot, MaybeNoteModifier.

3.2. Símbolos Terminales

#tempo, #compas, '/', const, const_name, figure, number, ';', voz, '′,′', '{', '}', compas, repetir, silencio, nota, note, '.', ',', note_modifier.

3.3. Producciones

Estas son las producciones utilizadas por nuestro parser para generar el AST de la cadena de entrada. En **negrita** se resaltan los símbolos *no terminales* para facilitar la lectura.

Se decidió requerir que exista al menos una voz y un compás por voz a nivel de la Gramática.

Start	→	**TempoDefinition BarDefinition Constants Voices**
TempoDefinition	→	#tempo figure number
BarDefinition	→	#compas number/number
Constants	→	λ | const const_name=number; **Constants**
Voices	→	**Voice MaybeVoices**
MaybeVoices	→	λ | **Voice MaybeVoices**
Voice	→	voz (number) bars | voz (const_name) **Bars**
Bars	→	**Bar MaybeBars**
MaybeBars	→	λ | **Bar MaybeBars**
Bar	→	**Repeat** | compas **Notes**
Repeat	→	repetir (number) bars | repetir (const_name) bars
Notes	→	λ | silencio (figure **MaybeDot**); **Notes** |
		note_call (note **MaybeNoteModifier**, number, figure **MaybeDot**); **Notes** |
		note_call (note **MaybeNoteModifier**, const_name, figure **MaybeDot**); **Notes**
MaybeDot	→	λ | .
MaybeNoteModifier	→	λ | note_modifier

3.4. Expresiones regulares de los tokens

El lado izquierdo de cada una de las siguientes líneas es el valor de un *token* dado, y el lado derecho es la expresión regular que reconoce las cadenas que luego se transforman en el mismo al hacer la primera pasada del código con el lexer.

note_modifier	=	+ | -
const_name	=	[a-zA-Z][a-zA-Z0-9]*
number	=	[0-9]+
figure	=	redonda | blanca | negra | corchea | semicorchea | fusa | semifusa
note	=	do | re | mi | fa | sol | la | si

4. Tests

Para comprobar el funcionamiento correcto de **Musileng**, se crearon varios casos de tests.

En el directorio *tests/* se encuentran los tests de casos dónde se debe devolver un error:

- Error de sintaxis cuando hay una cadena que no se puede tokenizar
- Error de sintaxis cuando una cadena no coincide con ninguna producción
- Errores de semántica: Instrumentos y octavas fuera de rango, tempos y repeticiones con valor 0, etc.

Y en el directorio *examples/* se encuentran tests de casos favorables, modificados de los ejemplos de la cátedra para incluir más detalles, por ejemplo, repeticiones anidadas.

Se pueden comprobar los tests corriendo: **./tests.sh**

5. Manual

El trabajo práctico fue desarrollado y probado con las siguientes versiones de las herramientas sugeridas:

- Versión de Python: 2.7.9
- Versión de ply: 3.6
- Versión de midicomp: 0.0.6

Modo de uso

Para correr el trabajo práctico se respetó el comando dado por la cátedra. Se ejecuta:
./musileng entrada.mus salida.txt

6. Conclusiones

El trabajo práctico nos resultó una manera efectiva de unir los conocimientos adquiridos a lo largo del cuatrimestre, ya que consigue abarcar una amplia cantidad de temas troncales vistos en la cursada, de forma práctica y palpable.

Es distinto leer la teoría y hacer los ejercicios de las gramáticas, parsers y demás, sin tener esa bajada a tierra de la relación real entre lo visto y las cosas que uno está acostumbrado en el día a día (código propiamente dicho).