Compilateur - TP

Cédric Pahud, Guillaume Noguera

INF3-dlma

Introduction

Pour ce TP, nous avons choisi de développer un compilateur musical basé sur la librairie cSound. Bien que flexible et puissante, cSound est une ancienne librairie qui possède une syntaxe plutôt compliquée : Avant d'écrire une partition, les instruments doivent d'abord être déclarés et leurs paramètres attribués. Côté partition, chaque note doit être déclarée "à la main", sa durée et son temps de départ devant être spécifié.

Notre but était donc de simplifier cette syntaxe, quitte à perdre un peu (beaucoup) de flexibilité.

Exemple cSound :

```
<CsoundSynthesizer>
# Options de sortie
<CsOptions>
-odac
-o out.wav _W
</CsOptions>

# Déclaration d'un instrument et binding des paramètres
<CsInstruments>
sr=44100
ksmps=10
nchnls=1
instr 1
iamp=p4
ifreq=cpspch(p5)
iatt=p6
idec=p7
islev=p8
irel=p9
kenv adsr p3/iatt, p3/idec, islev, p3/irel
aout oscil iamp*kenv, ifreq, 1
out aout
endin
</CsInstruments>

<CsScore>
# Déclaration de la forme d'onde
f1 0 16384 10 1 0.5 0.3 0.25 0.2 0.167 0.14 0.125 .111
# Début de la partition
i 1 1.50 1.50 10000.00 5.09 1.00 6.00 1.00 1.50
i 1 3.00 1.50 10000.00 6.09 1.00 6.00 1.00 1.50
i ...
```

Environnement

Pour utiliser notre compilateur, il est nécessaire d'installer CSound. La sortie de compliler.py donnera alors un fichier `.csd` et un fichier `.wav` car on appelle cSound depuis python grace à subprocess.

Analyseur lexical

Pour développer l'Analyseur lexical, nous nous sommes basé sur le fichier lex5.py développer dans les tp du cours de compilateur.

Afin de gérer les notes sans trop se compliquer la vie, nous avons utilisé une liste de notes à laquelle nous avons ajouté un chiffre de 1 à 9 correspondant à l'octave. Finalement, nous utilisons la fonction `join()` pour créer une regexp facilement grace au décorateur `@TOKEN` car on ne peut pas le faire directement dans la docstring.

```
notes = (
    'do',
    'do\#',
    're',
    're\#',
    'mi',
    'mi\#',
    'fa',
    'fa\#',
    'sol',
    'sol\#',
    'la',
    'la\#',
    'si',
```

```
    'si\#'
)

notes = [note+str(i) for note in notes for i in range(1,9)]

@TOKEN(r'|'.join(notes))
def t_NOTE(t):
    return t
```

Notre code ne possédant pas de marqueur de fin de ligne, nous avons créé un un token `t_NEWLINE` qui permet de reconnaitre une ou plusieurs fin de lignes.

```
def t_NEWLINE(t):
    r'\n+'
    t.lexer.lineno+=len(t.value)
    return t
```

Nous avons aussi utilisé un token `t_comment` permettant de reconnaitre les commentaires sur une ligne commençant par `#`. Par contre, les commentaires multilignes ne sont pas implémentés.

```
def t_comment(t):
    r'\#.*\n*'
    t.lexer.lineno+=t.value.count('\n')
```

Parseur

Pour le parseur,le code se sépare en deux partie. Les assignations et le code à proprement parlé. L règle programme ressemble donc à ça :

```
def p_programme(p):
    '''programme : assignationBlock START NEWLINE codeBlock STOP NEWLINE'''
    p[0]=AST.ProgramNode([p[1],AST.StartNode(),p[4],AST.StopNode()])
```

Pour l'assignation d'accord, sachant qu'une liste de notes peut être de taille variable nous avons utilisé un règle récursive permettant de gérer le nombre de notes que l'on veut pour un accord:

```
def p_notelist(p):
    '''
    notelist : notelist ',' NOTE
    notelist : NOTE
    '''
    if len(p) == 2:
        p[0] = [AST.NoteNode(p[1])]
    else:
        p[0] = p[1]
        p[0].append(AST.NoteNode(p[3]))
```

AST et compiler

Dans l'AST, nous avons ajouté beaucoups de noeud dont seulement le type change afin de simplifier l'utilisation de notre compiler. Le seul noeud intéressant est le noeud note qui stock la note et l'octave :

```
class NoteNode(Node):
    type = 'note'
    def __init__(self, note):
        Node.__init__(self)
        if note[2]=="#":
            self.note = note[:3]
        else:
            self.note = note[:2]
        self.hauteur = int(note[-1])

    def __repr__(self):
        return repr(self.note)+" "+repr(self.hauteur)
```

Pour ce qui est du compiler nous nous sommes beaucoup simplifié la vie avec le fichier `utils.py` qui à permis une très bonne abstraction du code. La plupart du code ajouté dans les noeuds se résume principalement à des if avec des appels aux fonctions du fichier `utils.py`

Syntaxe

Déclaration du BPM et des instruments :

```
BPM = 110
I_1 = square
I_2 = sine
```

Déclaration d'accords :

```
moican = [ do4, do6 ,mi4 ]
narval = [ re6, fa6, la6, do7, mi7]
```

Déclaration du début de la partition

```
  START
```

Déclaration d'une boucle for :

```
    REP(2)
    {
    I_1 do4
    I_2 mi5
    # permet de définir une pause d'un temps pour un instrument
    I_1 PAUSE
    I_1 fa6
    I_2 moican
    I_1 narval
    # Les accords peuvent également être déclarés inline
    I_2 [ do5, mi5 ]
    }
```

Déclaration d'un arpège :

```
# deux octaves, montant :
ARP(I_1,moican,2,+)
#inline, descendant
ARP(I_2,[do4, do6, mi4],2,-)
```

Les instruments doivent être impérativement déclarés avant de commencer une partition, afin de respecter le fonctionnement de cSound (il ne s'agit là que d'un wrapper, en somme). A noter que les commentaires commençant par un `#` sont géré par notre compilateur

Partie métier

Le fichier utils.py s'occupe de la génération du code cSound. Toutes les méthodes concernant une génération de strings sont précédées du sucre syntaxique "d_"

Il s'occupe de gérer les instruments ainsi que leurs offsets, notre compilateur fonctionnant en mode "note à note" (cSound attendant de nous de spécifier le temps de départ de chaque note):

```
class Instrument():
    def __init__(self, id, type):
        self.id = id
        self.type = type
        self.offset = 0
```

à chaque fois qu'un instrument joue une note, son offset est augmenté.

Les classes suivantes sont également présente pour stocker les réglages, notes et accords déclarés :

```
class Settings():
    def __init__(self):
        self.bpm = 120
        self.dt = 0.5

    def set_bpm(self, bpm):
        self.bpm = bpm
        self.dt = 60/self.bpm

class Note():
    def __init__(self, key, octave):
        self.key = key
        self.octave = octave

class Chord():
    def __init__(self, notes):
        self.notes = notes
```

les fonctions principales sont les suivantes :

```

''' Fonctionnel '''
# Ajoute un instrument. Types disponibles : sine, square, saw, pulse
def add_instr(name, type):
  # ...

# Force un instrument à faire une pause spécifiée par le paramètre dur
def wait(instrName, dur):
```

```
  # ...

''' Génération de string '''
# Retourne un string comportant la déclaration de base d'une partition cSound.
def d_starting():

# Retourne un string comportant la déclaration de tous les
# instruments précédemments déclarés, ainsi que leur type.
# Doit être appellée au début du fichier.
def d_instruments():
  # ...

# Retourne un string comportant la déclaration d'une note unique.
# Les paramètres a, d, s, r ne sont pour l'instant
# pas utilisés mais définissent la forme d'onde d'une note.
# le paramètre chord est utilisé pour évité l'incrémentation
# de l'offset d'un instrument lors de la création d'accords.
def d_note(instrName, dur, amp, note, a, d, s, r, sta=None, chord=False):
  # ...

# Déclare un accord, soit n notes déclenchées au même moment par un
# même instrument.
def d_chord(instrName, dur, amp, chord, a, d, s, r, sta=None):
  # ...

# Retoure un string déclarant un arpège : Répète un accord sur
# plusieurs octaves avec une durée de note constante.
# Le sens (montant / descendant) peut être spécifié par la variable inc (-1 / 1)
def d_arp(instrName, dur, amp, chord, a, d, s, r, loops, inc, sta = None):
  # ...

# Retourne le string terminant la déclaration d'une partition cSound.
def d_end():
  #...
```