[WIP] Relatório de Pré-Dissertação Mestrado em Engenharia Informática

Pedro Miguel Oliveira da Silva

Setembro, 2019

1 Sinopse

Candidato	Pedro Miguel Oliveira da Silva
Tema	DSL para programação de teclados e acompanhamentos musicais dinâmicos virtuais
Orientação	José João Almeida
Instituição	Departamento de Informática Escola de Engenharia Universidade do Minho

2 SoundFonts

O formato *SoundFont* foi originalmente desenvolvido nos anos 90 pela empresa E-mu Systems para ser usado inicialmente pelas placas de som Sound Blaster. Ao longo dos anos o formato sofreu diversas alterações, encontrando-se atualemente na versão 2.04, lançada em 2005[1]. Atualmente existem diversos sintetizadores de software *cross platform* e open source capazes de converterem eventos *MIDI* em som usando ficheiros SoundFont, dispensando a necessidade de uma placa de som compatível com o formato. Alguns destes projetos são TiMidity++, WildMIDI e FluidSynth.

Um ficheiro de SoundFont é constítuido por um ou mais bancos (*banks*) (até um máximo de 128). Cada banco pode por sua vez ter até 128 *presets* (por vezes também chamados instrumentos ou programas).

TODO

3 FluidSynth

A biblioteca FluidSynth é um *software* sintetizador de aúdio em tempo real que transforma dados MIDI em sons, que podem ser gravados em disco ou encaminhados diretamente para um *output* de aúdio. Os sons são gerados com recurso a SoundFonts[1] (ficheiros com a extensão `.sf2`) que mapeiam cada nota para a gravação de um instrumento a tocar essa nota.

Os *bindings* da biblioteca para C# foram baseados no código *open source* do projeto NFluidSynth[2], com algumas modificações para compilar com a versão da biblioteca em Linux.

3.1 Inicialização

Para utilizar a biblioteca FluidSynth, existem três objetos principais que devem ser criados: Settings (`fluid_settings_t*`), Synth (`fluid_synth_t*`) e AudioDriver (`fluid_audio_driver_t*`).

O objecto **Settings**[3] é implementado com recurso a um dicionário. Para cada

chave (por exemplo, "`audio.driver`") é possível associar um valor do tipo inteiro (`int`), *string* (`str`) ou *double* (`num`). Alguns valores podem ser também booleanos (`bool`), no entanto eles são armazenados como inteiros com os valores aceites sendo apenas 0 e 1.

O objeto **Synth** é utilizado para controlar o sintetizador e produzir os sons. Para isso é possível enviar as mensagens MIDI tais como `NoteOn`, `NoteOff`, `ProgramChange`, entre outros.

O terceiro objeto **AudioDriver** encaminha automaticamente os sons para algum *audio output*, seja ele colunas no computador ou um ficheiro em disco. Os seguintes *outputs* são suportados pela biblioteca:

Linux: jack, alsa, oss, PulseAudio, portaudio, sdl2, file

Windows: jack, PulseAudio, dsound, portaudio, sdl2, file

Max OS: jack, PulseAudio, coreaudio, portaudio, sndman, sdl2, file

Android: opensles, oboe, file

3.2 Utilização

Com os objetos necessários inicializados, é necessário ainda especificar qual (ou quais) a(s) *SoundFont(s)* a utilizar. Para isso podemos chamar o método `Synth.LoadSoundFont` que recebe dois argumentos: uma *string* com o caminho em disco do ficheiro *SoundFont* a carregar, seguido dum booleano que indica se os *presets* devem ser atualizados para os da nova *SoundFont* (isto é, atribuir os instrumentos da *SoundFont* aos canais automaticamente).

A função `Synth.NoteOn` recebe três argumentos: um inteiro a representar o canal, outro inteiro entre 0 e 127 a representar a nota, e finalmente outro inteiro também entre 0 e 127 a representar a velocidade da nota.

O canal (**channel**) representa qual o instrumento que vai reproduzir a nota em questão. Cada canal está atríbuido a um programa da SoundFont, e é possível a qualquer momento mudar o programa atribuido a qualquer canal através do método `Synth.ProgramChange`. Caso se tenha carregado mais do que uma *SoundFont*, é possível usar o método `Synth.ProgramSelect`, que permite especificar o `id` da *SoundFont* e do banco do instrumento a atribuir.

A chave (**key**) representa a nota a tocar. Sendo este valor um inteiro entre 0 e 127, é necessário saber como mapear as tradicionais notas músicais neste valor. Para isso, basta colocarmos as *pich classes* e os seus respetivos acidentais *sharp* numa lista ordenada (`C, C#, D, D#, E, F, F#, G, G#, A, A#, B`) e associar a eles os inteiros entre 0 e 11 (inclusive). Depois apenas temos de somar a esse número a multiplicação da oitava da nota (a começar em 0) por 12. Podemos deste modo calcular, por exemplo, que a *key* do C central (C4) é igual a 48 ($0 + 4 * 12$).

$$N + O * 12$$

A velocidade (**velocity**) é também um valor entre 0 e 127. Relacionando a velocidade com um piano físico, esta representa a força (ou velocidade) com que a tecla foi premida. Velocidades maiores geram sons mais altos, enquanto que velocidades mais baixas geram sons mais baixos, permitindo assim ao músico dar ou tirar enfase a uma nota relativamente às restantes. De notar que um valor igual a zero é o equivalente a invocar o método `Synth.NoteOff`.

A método `Synth.NoteOff`, por sua vez, recebe apenas dois argumentos (canal e chave), e deve ser chamada passsado algum tempo para terminar a nota. Podemos deste modo construir a analogia óbvia que o método `NoteOn` corresponde a uma tecla de piano ser premida, e `NoteOff` corresponde a essa tecla ser libertada.

4 Gramáticas

Para além dos aspetos técnicos da geração e reprodução de música já abordados neste relatório, existe também um componente fulcral relativo à análise e interpretação da linguagem que irá controlar a geração dos sons. Uma das primeiras decisões a ser tomada diz respeito à escolha do *parser*, e possivelmente, do tipo de gramática que irá servir de base para a geração do mesmo.

Tradicionalmente, as gramáticas mais populares no campo de processamento de texto tendem a ser Context Free Grammar (CFG), que são usadas como *input* nos geradores de *parser* mais populares (Bison/YACC, ANTLR). Existem no entanto alternativas, algumas mais recentes, como as Parsing Expression Grammar (PEG), que trazem consigo diferenças que podem ser consideradas por alguns como vantagens ou desvantagens.

4.1 Diferenças: CFG vs PEG

A diferença com maiores repercussões práticas entre as duas classes de gramáticas deve-se á semãntica atribuída ao operador de escolha, e a consequente **ambíguidade** (ou falta dela) na gramática. Nas gramáticas PEG, o operador é ordenado, o que significa que a ordem porque as alternativas aparecem é relevante durante o *parse* do *input*. Isto contrasta com a semântica nas CFG, onde a ordem das alternativas é irrelevante. Isto pode no entanto levar a ambíguidades, onde o mesmo *input*, descrito pela mesma gramática, pode resultar em duas árvores de *parsing* diferentes. Isto é, as CFG podem por essa razão ser ambíguas.

Tomemos como exemplo o famoso problema do *dangling else*[4] descrito nas duas classe de gramáticas:

```
if (a) if (b) f1(); else f2();
```

Listing 1: Gramática

```
statement = ...
    | conditional_statement

conditional_statement = ...
    | IF ( expression ) statement ELSE statement
    | IF ( expression ) statement
```

No caso de uma CFG, sabendo que o operador de escolha | é comutativo, o seguinte *input* será ambíguo, podendo resultar num *if-else* dentro do *if* ou num *if* dentro de um *if-else*.

Mas no caso de uma PEG, o resultado é claro: um *if-else* dentro de um *if*. Quando a primeira regra do condicional chega ao `statement`, este vai por sua vez chamar o não terminal `conditional_statement`, que por sua vez irá consumir o *input* até ao fim. Deste modo, quando a execução voltar ao primeiro `conditional_statement`, esta irá falhar por não conseguir ler o *else* (uma vez que já consumimos todo o texto de entrada). Deste modo irá usar a segunda alternativa, dando então o resultado previso.

Com este exemplo de *backtracking* podemos também verificar um problema aparente nas gramáticas PEG. Falhando a primeira alternativa na produção `conditional_statement`, a segunda irá ser testada. Mas é evidente, olhando para a gramática que a segunda alternativa é exatamente igual à parte inicial da primeira alternativa (que neste caso também corresponde á parte que teve sucesso). Em vez de voltar a testar as regras de uma forma *naive*, as Parsing Expression Grammar guardam antes em *cache* os resultados de testes anteriores, permitindo assim uma pesquisa em tempo linear relativamente ao tamanho do *input*, à custa de uma maior utilização de memória.

4.1.1 Resumo

Em resumo, as três principais diferenças entre as tradicionais Context Free Grammar (CFG) e as mais recentes Parsing Expression Grammar (PEG) são:

Ambiguidade. O operador de escolha ser comutativo nas CFG resulta em gramáticas que podem ser ambíguas para o mesmo *input*. As PEG são determínisticas, mas exigem mais cuidado na ordem das produções, uma vez que tal afeta a semântica da gramática.

Memoization Para evitar *backtracking* exponêncial, as PEG utilizam *memoization* que lhes permite guardar em *cache* resultados parciais durante o processo de *parsing*. Isto reduz o tempo dispendido, pois evita fazer o *parse* do mesmo texto pela mesma regra duas vezes. Mas também aumenta o consumo de memória, pois os resultados parciais têm de ser guardados até a análise terminar por completo.

Composição As Parsing Expression Grammar também têm a vantagem de oferecerem uma maior facilidade de composição. Em qualquer parte da gramática é possível trocar um terminal por um não terminal. Isto é, é extremamente fácil construir gramáticas mais modulares e compô-las entre si.

4.2 Acompanhamentos Músicais

A gramática de expressões ou acompanhamentos músicais tem como base fundamental os seguintes blocos: notas, pausas e modificadores. As notas são identificadas pelas letras A até G, seguindo a notação de *Helmholtz*[5] para de-

notar as respetivas oitavas. Podem também ser seguidas de um número ou de uma fração, indicando a duração da nota.

Exemplos de notas

```
C,, C, C c c' c'' c''' c'/4 A1/4 B2
```

As notas podem depois ser compostas sequencialmente (como demonstrado em cima) ou em paralelo (separados por uma barra vertical |). Devemos notar que o operador paralelo tem a menor precedência de todos, pelo que não é necessário agrupar as notas com parênteses quando se usa. Isto é, as duas expressões seguintes são equivalentes.

```
A B C | D E F
( A B C ) | ( D E F )
```

É também possível agrupar estes blocos com recurso a parênteses. Os grupos herdam o contexto da expressão superior, mas as modificações ao seu contexto permanecem locais. Isto permite, por exemplo, modificar configurações para apenas um conjunto restrito de notas. No exemplo seguinte, a velocidade da nota `C` é 70, mas para o grupo de notas `A B` a velocidade é 127.

```
v70 (v127 A B) C
```

Os modificadores disponíveis são:

Velocity A velocidade das notas, tendo o formato `[vV][0-9]+`.

Duração A duração das notas, tendo o formato `[lL][0-9]+` ou `[lL][0-9]+/[0-9]+`.

Tempo O número de batidas por minuto (BPM) que definem a velocidade a que as notas são tocadas, tendo o formato `[tT][0-9]+`.

Assinatura de Tempo Define a assinatura de tempo, que define o tipo de batida da música e o comprimento de uma barra na pauta musical. Tem o formato `[sS][0-9]+/[0-9]`.

É também poossível definir qual o instrumento a ser utilizado para as notas. Todas as notas pertencentes ao mesmo contexto depois do modificador utilizarão esse instrumento.

```
(:cello A F | :violin A D)
```

Para além destas funcionalidades, também existe algum açúcar sintático para algumas das tarefas mais comuns na construção de acompanhamentos, como tocar acordes ou repetir padrões.

```
( [BG]*2 [B2G2] )*3
```

A gramática completa pode ser analizada no **Anexo A**.

References

[1] Soundfont technical specification. `http://www.synthfont.com/sfspec24.pdf`, February 2006.

[2] Atsushi Eno. Nfluidsynth. `https://github.com/atsushieno/nfluidsynth`, 2019.

[3] Fluidsynth settings. `http://www.fluidsynth.org/api/fluidsettings.xml`.

[4] Dangling else. `https://en.wikipedia.org/wiki/Dangling_else`.

[5] Helmholtz pitch notation. `https://en.wikipedia.org/wiki/Helmholtz_pitch_notation`.

Appendices

A Gramática

```
body = _ expression _

expression = parallel

parallel
    = sequence _ "|" _ parallel
    | sequence

sequence <MusicNode>
    = repeat _ sequence
    | repeat

repeat
    = expressionUnambiguous _ "*" _ integer
    | expressionUnambiguous

expressionUnambiguous
    = group | chord | note | rest | modifier | instrumentModifier

group
    = "(" _ expression _ ")"

note
    = notePitch _ noteValue
    | notePitch

chord = "[" _ chordBody  _ "]"

chordBody
    = note _ chordBody
    | note
```

```
rest
    = "r" _ noteValue
    | "r"

noteValue
    = "/" _ integer
    | integer _ "/" _ integer
    | integer

notePitch
    = [cdefgab] "'"*
    | [CDEFGAB] ","*

modifier
    = [tT] _ integer
    | [vV] _ integer
    | [lL] _ noteValue
    | [sS] _ integer _ "/" _ integer
    | [sS] _ integer
    | [oO] _ integer

instrumentModifier
    = ":" alphanumeric _ sequence

integer
    = [0-9]+

alphanumeric
    = [a-zA-Z][a-zA-Z0-9]*

_ = [ \t\r\n]*
```