

**Universidade do Minho**

Escola de Engenharia

Departamento de Informática

Pedro M. Silva

**Musikla**  
**Music and Keyboard Language**

**DSL de criação de acompanhamentos e  
e teclados musicais programáveis**

January 2020



**Universidade do Minho**

Escola de Engenharia

Departamento de Informática

Pedro M. Silva

**Musikla**  
**Music and Keyboard Language**

**DSL de criação de acompanhamentos e  
e teclados musicais programáveis**

Master dissertation  
Master Degree in Computer Science

Dissertation supervised by  
**José João Almeida**

January 2020

---

## ABSTRACT

---

The creation of music using synthesizers is a practice that boasts many decades of existence. With the proliferation of personal computers, digital synthesizers and Digital Audio Workstations rose in popularity as well.

This project aims to use the digital audio production and manipulation technologies, and wrap them in a domain specific language that allows to easily describe music compositions, generate sounds dynamically, study properties of music theory or even create virtual keyboards that play sounds or execute actions and can be used to perform live multimedia shows.

---

## RESUMO

---

A criação de música usando sintetizadores é uma prática que conta já com várias décadas de uso. Com a proliferação dos computadores pessoais, popularizaram-se também os sintetizadores digitais e as Digital Audio Workstations.

Este projeto tem como objetivo utilizar as tecnologias de produção e manipulação de áudio digital, e envolvê-las numa linguagem de domínio específico que permita facilmente descrever composições musicais, gerar sons dinâmicamente, estudar propriedades da própria teoria musical, ou até criar teclados virtuais que toquem sons ou executem ações e podem ser utilizados para realizar espetáculos multimédia ao vivo.

---

## CONTENTS

---

<b>1</b>	<b>INTRODUÇÃO</b>	<b>1</b>
<b>2</b>	<b>ESTADO DA ARTE</b>	<b>2</b>
2.1	Trabalho Relacionado	2
2.1.1	Alda	2
2.1.2	ABC Notation	3
2.1.3	Faust	4
2.1.4	Sonic Pi	6
2.2	Gramáticas	7
2.2.1	Diferenças: CFG vs PEG	8
2.2.2	Resumo	9
2.3	SoundFonts	9
2.4	Sintetizadores	11
2.4.1	Inicialização	11
2.4.2	Utilização	12
<b>3</b>	<b>O PROBLEMA E OS SEUS DESAFIOS</b>	<b>14</b>
3.1	Solução Proposta	14
3.1.1	Gramática da Linguagem	16
3.1.2	Arquitetura do Sistema	20
<b>4</b>	<b>MUSIKLA: CASOS DE ESTUDO</b>	<b>23</b>
4.1	Tocar Música	23
4.2	Definir um teclado	24
4.3	QWERTY Keyboard	25
<b>5</b>	<b>PLANEAMENTO</b>	<b>28</b>
<b>6</b>	<b>CONCLUSÃO</b>	<b>29</b>
<b>A</b>	<b>APPENDIX</b>	<b>31</b>
A.1	Gramática	31

---

## LIST OF FIGURES

---

Figure 1	Arquitectura Geral do Projeto	20
Figure 2	Arquitectura do Interpretador	21
Figure 3	Calenderização Prevista	28

---

LIST OF TABLES

---

# 1

---

## INTRODUÇÃO

---

O objetivo deste trabalho é estudar e prototipar formas de criação de música com recurso a técnicas habitualmente usadas na criação de *software*. Para além de permitir criar música através das notas introduzidas manualmente, a linguagem deve facilitar a geração de música de um modo mais dinâmico, com recurso a operações de combinação e transformação de notas, como concatenar e misturar música.

O termo música neste contexto é usado num sentido mais amplo que apenas sons. O objetivo desta linguagem é permitir gerar vários *outputs* através do mesmo código fonte, como pautas musicais, ABC, WAV, MIDI, entre outros.

Uma das partes mais críticas relativas à pesquisa e desenvolvimento necessários para a realização desta linguagem é a componente temporal implícita em todos os aspetos da linguagem: deve ser possível de um modo intuitivo expressar as várias composições possíveis de notas sem ser necessário expressar os tempos manualmente, tais como notas sequênciais, notas em paralelo, acordes, pequenas pausas e grandes pausas, bem como sincronizar partes da música de modo simples.

# 2

---

## ESTADO DA ARTE

---

Atualmente a produção de música é realizada utilizando programas com interfaces gráficas, geralmente denominados como [Digital Audio Workstation \(DAW\)](#). A minha abordagem irá consistir em estudar formas de criar e tocar músicas ao vivo (e não só) através de uma [Domain Specific Language \(DSL\)](#), usando técnicas inspiradas nas linguagens de programação e no desenvolvimento de *software*.

### 2.1 TRABALHO RELACIONADO

Existem diversos tipos de linguagens usadas atualmente para produzir ou simplesmente descrever música. Algumas fazem uso do conceito de notas musicais, com recurso a algum sintetizador externo, para gerar os sons, enquanto outras funcionam com base na manipulação direta de ondas de som digitais para criar música. Algumas suportam apenas a descrição estática da música, enquanto outras permitem formas dinâmicas tais como funções, variáveis, estruturas de controlo e repetição, ou até mesmo algorítmos aleatórios que permitem gerar músicas diferentes a cada execução.

Iremos de seguida analisar algumas das soluções disponíveis atualmente, bem como comparar as funcionalidades que cada uma oferece ou não oferece em relação aos objetivos deste projeto.

#### 2.1.1 *Alda*

O projeto [alda](#)[1] é uma linguagem de música textual desenvolvida em JAVA focada na simplicidade: o seu maior ponto de atração é apelar tanto a programadores com pouca experiência musical, bem como a músicos com pouca experiência com programação. Apesar de ser anunciada como direcionada tanto a músicos como a programadores, a linguagem não suporta nenhum tipo de construções dinâmicas, como ciclos ou funções. Este tipo de funcionalidades, se necessário, requer o uso de uma linguagem de programação por cima, que poderia por exemplo, gerar o código *alda* em *runtime* através da manipulação de *strings* antes de o executar. Isto significa que não é possível implementar composições interativas.

### *Exemplos*

O exemplo seguinte demonstra um simples programa escrito em *alda*, demonstrando: a seleção de um instrumento (*piano*:), a definição da oitava base (o3), um acorde com quatro notas (c1/e/g/>c4) em que a última se encontra uma oitava acima das outras.

```
piano: o3 c1/e/g/>c4 < b a g | < g+1/b/>e
```

Listing 2.1: Exemplo da linguagem alda

É também possível verificar o uso de acidentais (identificados pelos símbolos + ou - a seguir a uma nota) bem como a diferenciação da duração de algumas notas (identificadas pelos números em frente às notas).

#### 2.1.2 ABC Notation

A notação **ABC**[2] é uma notação textual que permite descrever notação musical. É bastante completa, tendo formas de descrever notas, acordes, acidentais, uniões de notas, *lyrics*, múltiplas vozes, entre outros.

Para além das exaustividade de sintaxe que permite descrever quase todo o tipo de música, a popularidade da linguagem também significa que existem já inúmeros conversores de ficheiros ABC para os mais diversos formatos, desde ficheiros MIDI, pautas musicais, ou mesmo ficheiros WAV (gerados através do fornecimento de um ficheiro SoundFont, por exemplo).

A complexidade da notação traz tanto vantagens como desvantagens, no entanto: A sua ubiquidade significa que uma maior percentagem de utilizadores já se pode sentir à vontade com a sintaxe, o que não acontece com outras linguagens menos conhecidas. Mas por outro lado, conhecer ou implementar toda a especificação [3] é um feito bastante difícil.

No entanto, tal como a linguagem *ALDA*, as músicas definidas são estáticas, pelo que não serve como uma linguagem de programação de músicas dinâmicas. Ainda assim, apesar de implementar toda a notação ser algo pouco prático, implementar um *subset* da notação, contendo as construções mais usadas seria uma vantagem enorme que me permitiria aproveitar a familiaridade de muitos utilizadores com as partes mais comuns da sintaxe.

### *Exemplos*

A sintaxe de um ficheiro ABC é composta por duas partes: um cabeçalho onde são definidas as configurações da música atual, seguido pelo corpo da música. O cabeçalho é formado por uma várias linhas. Cada linha, em ABC chamada de campo, tem uma chave e um valor separados por dois pontos (:). A especificação da notação descreve bastantes campos

possíveis, mas os mais usados são: **X** (número de referência), **T** (título), **M** (compasso), **L** (unidade de duração de nota) e **K** (clave).

```
C, D, E, F, |G, A, B, C|D E F G|A B c d|e f g a|b c' d' e'|f' g' a' b'|]
```

Listing 2.2: Exemplo da notação ABC

No exemplo acima podemos ver uma escala completa das notas (sem acidentais). O chamado C médio é representado por um c minúsculo (a capitalização das letras muda o significado). Para subir uma oitava, podemos anotar as notas com um apóstrofo (c'). As oitavas subsequentes são anotadas por mais apóstrofos. De modo análogo, para baixar uma oitava, devemos usar primeiro a nota em maiúscula (C). As oitavas anteriores são identificadas por uma (ou mais) vírgula a seguir à nota com letra maiúscula (C,).

```
A/2 A/ A A2 __A _A =A ^A ^^A [CEGc] [C2G2] [CE][DF]
```

Listing 2.3: Exemplo da notação ABC

A duração das notas pode ser ajustada relativamente à unidade global definida no cabeçalho acrescentando um número (por exemplo 2) ou fração 1/4 à nota. Os acidentais bemol, bequadro e sustenido podem ser adicionados acrescentando um \_, = e ^ antes da nota, respetivamente. Acordes (notas tocadas ao mesmo tempo) podem ser definidas entre parênteses retos ([ e ]).

A notação disponibiliza muitos mais exemplos de todas as funcionalidades aceites no seu website[4].

### 2.1.3 Faust

A linguagem **Faust**[5] é uma linguagem de programação funcional com foco na sintetização de som e processamento de audio. Ao contrário das linguagens analisadas até agora, não trabalha com abstrações de notas e elementos musicais. Em vez disso, a linguagem trabalha diretamente com ondas sonoras (representadas como *streams* de números) e através de expressões matemáticas, que de uma forma funcional permite assim manipular o som produzido.

Um dos pontos fortes da linguagem é o facto da sua arquitetura ser construída de raiz para compilar o mesmo código fonte em várias linguagens. De facto, o projeto conta com várias dezenas de *targets*, desde os mais óbvios (C, C++, Java, JavaScript) até alguns mais especializados (WebAssembly, LLVM Bitcode, instrumentos VST/VSTi). Também permite gerar aplicações *standalone* para as bibliotecas de audio mais comuns[6].

A linguagem vem embutida com uma biblioteca extremamente completa[7] que implementa, entre muitas outras, funções de matemática comuns, filtros audio, funcionalidades extremamente básicas de interfaces gráficas que permitem controlar em tempo real os valores do programa (como botões e *sliders* entre outros).

### *Exemplos*

A documentação do projeto conta com uma quantidade abundante de exemplos[8] e com um tutorial para iniciantes à [9], do qual irei colocar aqui alguns pequenos pedaços de código que demonstram as capacidades fundamentais da linguagem.

```
import("stdfaust.lib");
process = no.noise*0.5;
```

Listing 2.4: Geração de ruído aleatório com volume a metade

No primeiro exemplo, podemos ver a estrutura mais básica de um programa escrito em *Faust*. Na primeira linha é importada a biblioteca *standard* da linguagem. Na segunda linha podemos ver a keyword **process**, que representa o *input* e *output* audio do nosso programa. Finalmente, em frente a essa keyword podemos ver a expressão `no.noise*0.5`. Isto demonstra a utilização de construções da biblioteca *standard*, como o gerador de ruído aleatório, bem como a utilização de operadores matemáticos usuais (neste caso a multiplicação) para manipular o audio, e diminuir o volume para metade.

```
import("stdfaust.lib");
ctFreq = 500;
q = 5;
gain = 1;
process = no.noise : fi.resonlp(ctFreq,q,gain);
```

Listing 2.5: Geração de ruído aleatório com um filtro *low-pass*

Neste exemplo, estamos a usar o operador `:` para canalizar o *output* do gerador de ruído para um filtro *low-pass*, que filtra todas as frequências acima de um valor de corte (a variável `ctFreq`). Aumentar esta variável resulta num som mais agudo, enquanto que ao diminui-la obtemos um som mais grave (pois o valor de corte é mais baixo, apenas os sons abaixo desse valor são passados).

```
import("stdfaust.lib");
ctFreq = hslider("[0]cutoffFrequency",500,50,10000,0.01);
q = hslider("[1]q",5,1,30,0.1);
```

```

gain = hslider("[2]gain",1,0,1,0.01);
t = button("[3]gate");
process = no.noise : fi.resonlp(ctFreq,q,gain)*t;

```

Listing 2.6: Geração de ruído aleatório com um filtro *low-pass* controlada por uma interface

Por fim podemos ver um exemplo igual ao anterior, mas em vez de ter os valores das variáveis estáticos (guardados nas variáveis `ctFreq`, `q` e `gain`), estes são controlados em tempo real pela interface definida pelas chamadas à função `hslider`. Foi também adicionada uma variável `t` com um botão "`gate`". Este produz o valor 0 (zero) quando está solto, e o valor 1 (um) quando está pressionado, valor que quando multiplicado pelo resto da expressão serve efetivamente como um *on-off switch* para todo o sistema.

#### 2.1.4 Sonic Pi

Possivelmente a linguagem que mais se aproxima do objetivo pretendido com este projeto, **Sonic Pi**[[10](#)] descreve-se como uma ferramenta de código para a criação e performance de música.

A linguagem permite tocar notas (e também construções mais complexas a partir das mesmas, tais como acordes, arpépios e escalas, por exemplo). Para além disso permite tocar *samples*, que são ficheiros **Waveform Audio File Format (WAV)**. A linguagem traz já consigo aproximadamente 164 *samples* que podem ser livremente usadas, mas é também possível ao utilizador usar as suas.

As músicas são compostas por **live loops**, que são grupos de sons. É possível ter vários *live loops* a tocar simultaneamente. Dentro de cada *live loop* o utilizador pode usar a função `play` para tocar notas, `sample` para reproduzir ficheiros **WAV**, ou `sleep` para avançar o tempo. Para além disso a linguagem suporta, através da função `with_fx` a reprodução de sons com efeitos (como *reverb*, *pan*, *echo* entre muitos outros[[11](#)]).

Para além das capacidades musicais, a linguagem disponibiliza numa sintaxe similar a *Ruby*, com construções de programação como ciclos, variáveis, estruturas de controlo, e até métodos para adicionar aleatoriedade à música tocada, permitindo escolher, por exemplo, qual a nota a tocar a seguir de uma lista de possibilidades.

Apesar de todas estas funcionalidades disponibilizadas, existem áreas onde o *Sonic Pi* fica aquém dos objetivos pretendidos para este projeto. Por exemplo, apesar de permitir tanto receber como enviar eventos **MIDI**, as suas capacidades de **Input/Output (I/O)** são bastante primitivas. Também não é fácil utilizar o teclado do computador para tocar sons ou manipular o estado do programa. É possível fazê-lo, mas é bastante mais complicado do que seria de esperar (para além de exigir utilizar alguma linguagem de programação à parte).

### Exemplos

```
loop do
    sample :perc_bell, rate: (rrand 0.125, 1.5)
    sleep rrand(0, 2)
end
```

Listing 2.7: Reproduzir um *sample* com valores aleatórios

Neste exemplo, podemos ver como a linguagem *Sonic Pi* permite criar um *loop*, onde podemos tocar sons (neste caso, um *sample* pré-definido chamado *perc\_bell*).

É possível verificar também o uso da função *sleep* para gerir manualmente o avanço temporal da música (neste caso usando um valor escolhido aleatoriamente e entre 0 e 2 segundos).

```
with_fx :reverb, mix: 0.2 do
    loop do
        play scale(:Eb2, :major_pentatonic, num_octaves: 3).choose, release: 0.1, amp: rand
        sleep 0.1
    end
end
```

Listing 2.8: Reproduzir um notas de uma escala aleatórias, com efeito *reverb*

Neste exemplo podemos observar a possibilidade do uso de efeitos, em particular do efeito *reverb*, para manipular o som gerado pelo programa. Dentro do *loop*, é tocada uma nota a cada 100 milisegundos. A função *scale* gera uma lista com as notas da escala pedida, e a função *choose* escolhe aleatoriamente uma dessas notas para tocar.

## 2.2 GRAMÁTICAS

Para além dos aspectos 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 até mais recentes, como as **Parsing Expression Grammar (PEG)**, que trazem consigo diferenças que podem ser consideradas por alguns como vantagens ou desvantagens.

### 2.2.1 Diferenças: CFG vs PEG

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

Tomemos como exemplo o famoso problema do *dangling else*[12] descrito nas duas classes de gramáticas:

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

Listing 2.9: 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). Irá depois irá usar a segunda alternativa, dando então o resultado previsto.

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 à 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.

### 2.2.2 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 determinísticas, mas exigem mais cuidado na ordem das produções, uma vez que tal afeta a semântica da gramática.

**Memoization** Para evitar *backtracking* exponencial, 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.

## 2.3 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 atualmente na versão 2.04, lançada em 2005[13]. 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++[14], WildMIDI[15] e FluidSynth[16].

Para além do formato original, existem também alternativas mais recentes que disponibilizam mais funcionalidades na sua especificação, como os ficheiros **SFZ** ou **NKI**. Estas alternativas trazem consigo vantagens e desvantagens, mas independentemente dos seus méritos, até agora nenhuma atingiu a popularidade dos ficheiros *SoundFont*, o que significa também menos bibliotecas e menos aplicações para trabalhar com elas.

Um ficheiro de *SoundFont* é constituído 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).

Usando a sintaxe de declaração de tipos do *Python* (que irá ser usada mais vezes neste projeto), podemos declarar um *SoundFont*, de uma forma bastante genérica e omitindo detalhes não essenciais, como sendo modelado pelos seguintes tipos:

```

# Cada preset é identificado por um par: (bank, preset number)
SoundFont = Dict[ Tuple[ int, int ], Preset ]

# Cada instrumento é identificado por um inteiro entre 0 e 127
Preset = Dict[ int, Instrument ]

# Finalmente, cada sample é identificada pelo índice de nota (que iremos abordar mais a
# frente em detalhe, no capítulo sobre sintetizadores)
Instrument = Dict[ int, Sample ]

# Aqui não se encontram detalhados os tipos Wave nem SampleOptions.
Sample = Tuple[ Wave, SampleOptions ]

```

Listing 2.10: Sistema de Tipos de um ficheiro SoundFont

## 2.4 SINTETIZADORES

A biblioteca FluidSynth é um *software* sintetizador de áudio em tempo real que transforma dados MIDI em sons, que podem ser gravados em disco ou encaminhados diretamente para um *output* de áudio. Os sons são gerados com recurso a SoundFonts[13] (ficheiros com a extensão .sf2) que mapeiam cada nota para a gravação de um instrumento a tocar essa nota.

Os *bindings* da biblioteca para Python foram baseados no código *open source* do projeto [pyfluidsynth](#)[17], jutamente com algumas definições CPython extra para permitir usar funções que não tivessem *bindings* já criados.

### 2.4.1 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 objeto **Settings**[18] 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 o 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

#### 2.4.2 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 atribuído a um programa da *SoundFont*, e é possível a qualquer momento mudar o programa atribuído 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 musicais neste valor. Para isso, basta colocarmos as *pitch 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 ( $0 + 4 * 12$ ). Assim, podemos generalizar que para uma oitava *O* e para um tom de nota *N*, obtemos a chave aplicando a fórmula:

$$N + O * 12$$

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 passado 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.

# 3

---

## O PROBLEMA E OS SEUS DESAFIOS

---

Desenhar esta *Domain Specific Language (DSL)* trás consigo os problemas comuns ao desenho de linguagens de programação, bem como desafios novos e únicos relativos ao domínio musical. Alguns desses desafios foram já bastante estudado pela míriade de linguagens de programação, tanto industriais como académicas, que já foram desenvolvidas, pelo que não serão o foco principal deste projeto. Pelo contrário, neste projeto serão focados com mais detalhe os desafios resultantes da integração da componente musical na linguagem.

O primeiro desses desafios é a introdução de um novo tipo de dados primitivo não existente na maioria das outras linguagens: **Música**. Este tipo de dados trás consigo a necessidade implícita de gerir o conceito de **tempo** na linguagem, tanto na geração de música *realtime* como *offline* (em que o tempo a que a música está a ser gerada pode ser mais rápido ou mais lento do que o tempo real). Este conceito de tempo também acaba por escapar para o campo da gramática e da sintaxe da linguagem, necessitando de uma forma de descrição do mesmo que seja flexível, mas não demasiado verbosa ou difícil de ler.

Ainda relacionado com o tipo de dados *Música*, também é importante pensar em como o representar, e os casos que deve cobrir. Para este fim, acho que é importante a linguagem permitir gerar sons **potencialmente infinitos**. Esta funcionalidade não é tão útil no campo da geração de música *offline*, mas é extremamente útil quando a música está a ser gerada em tempo real, e possivelmente a ser controlada por um utilizador através do teclado, permitindo começar a tocar música gerada proceduralmente, e deixá-la tocar durante o tempo que for necessário. Como tal é necessário pensar em como a implementação de todo o código depende deste ponto.

### 3.1 SOLUÇÃO PROPOSTA

A linguagem irá ser desenvolvida em *Python*. A linguagem irá ser extensível, permitindo ao utilizador definir objetos ou funções em *Python* e expô-los para dentro da linguagem, dando assim acesso à grande quantidade de módulos já existentes para os mais variados fins.

Como exemplo da extensibilidade da linguagem, irá também ser desenvolvido por cima dela uma biblioteca de construção de teclados virtuais que permitem associar a eventos de

teclas notas ou sequências musicais, ou mesmo instruções a serem executadas na própria linguagem.

Para resolver o problema da representação do tempo, toda a linguagem irá ter noção implícita desse conceito, mesmo que apenas algumas construções o utilizem. Isto significa que durante toda a execução, haverá uma variável de **contexto** que será implicitamente passada para todas instruções e todas as chamadas de funções que, entre outras coisas, irá manter registo da passagem do tempo. Desta forma os construtores que precisarem do contexto, como por exemplo a emissão de notas musicais, podem aceder ao tempo atual bem como modificá-lo.

A existência deste **contexto** implícito significa que as funções *Python* não podem ser expostas diretamente para a linguagem, mas graças à expressividade do *Python* é possível construir uma *Foreign Function Interface* que seja incrivelmente simples de usar e que evite que o utilizador tenha de mapear as funções manualmente. Em vez disso, pode simplesmente marcá-las como sendo **context-free** (funções que não têm noção da existência do contexto implícito), e elas serão então tratadas de forma apropriada.

O tipo de dados **Música** irá ser implementado sobre o conceito de iteradores (e mais especificamente geradores) fornecido pelo *Python* para tornar a criação de música *lazy*. No entanto, este paradigma deve ser completamente opaco para o utilizador da linguagem: a decisão de usar o modelo de execução normal, ou funções geradores deve ser tomado em segundo plano pelo motor de execução da linguagem, sempre que este for necessário. Isto é, ao contrário da maioria das linguagens que exigem para a utilização de geradores que o utilizador declare explicitamente que quer "emitir" um valor através de alguma *keyword*, geralmente *yield*, na nossa linguagem sempre que alguma função produzir um valor do tipo de música que não seja consumido de alguma forma (atribuído a uma variável ou passado a uma função, por exemplo), esse valor musical é **implicitamente emitido** para o gerador, uma vez que esse caso será o mais comum. Para evitar que o valor seja emitido, é necessário **descartá-lo manualmente** onde for caso disso. Se por outro lado a função lidar apenas com valores não-musicais, a sua execução irá seguir o modelo tradicional (onde a função termina a sua execução antes de retornar o controlo ao local onde foi chamada).

### 3.1.1 Gramática da Linguagem

A gramática completa da linguagem pode ser vista em A.1. MAs antes de abordarmos em mais detalhe como irá ser desenhada a gramática da linguagem, podemos aborar dois pequenos exemplos que demonstram a geração de notas musicais.

```
V70 L1 T120;

fun melody () {
    V120;

    r/4 ^g/4 ^g/4 ^g/4;
    ^f/2 e/8 ^d3/8;
    ^c2;
}

fun accomp () {
    V50; sustainoff();

    ^Cm;
    BM;
    AM;
}

# Create the notes from a melody with a piano and accompany it with a violin in parallel
$notes = :piano melody() | :violin accomp();

# Play the notes twice
play( $notes * 2 );
```

Listing 3.1: Exemplo da sintaxe proposta da linguagem

O desenho da gramática da linguagem é composto aproximadamente por três partes, todas elas interligadas entre si.

**INSTRUÇÕES E DECLARAÇÕES** Similar a quase todas as linguagens de programação, esta parte cobre a declaração de funções, variáveis, operadores e expressões em geral.

**ACOMPANHAMENTOS MÚSICAIS** Um tipo de expressões especial, que em vez de produzir números ou *strings* literais, reconhece expressões musicais (notas, acordes, etc).

**TECLADOS VIRTUAIS** Açúcar sintático para facilitar a declaração de teclados virtuais. Para além de alguns construtores próprios, faz uso das expressões gerais e de acompanhamentos musicais descritas acima.

### *Instruções e Expressões*

As instruções e expressões da gramática são baseadas nas linguagens como C e JavaScript. Chavetas são usadas para delinear blocos de código. As variáveis são prefixadas com um dólar (\$) para prevenir ambiguidades com notas musicais. Cada instrução é separada com um ponto e vírgula (;) a menos que sejam blocos de código que terminem com o fechar de chavetas. Os parâmetros de funções são separados por vírgulas, mas como as vírgulas também são usadas para indicar a descida de oitava nas notas, o ponto e vírgula (;) pode ser usado nesses casos para prevenir ambiguidades.

Em termos de instruções suportadas, a linguagem irá ter as usuais:

```
DECLARAÇÃO DE FUNÇÕES fun function_name ( $arg1, $arg2, <...> ) { }

CICLOS WHILE while <condition> { }

CICLOS FOR for $var in <expr> { }

CONDICIONAIS IF if <condition> { } else { }

ATRIBUIÇÕES A VARIÁVEIS $var = <expr>;

CHAMADA DE FUNÇÕES function_name( <arg1>, <arg2>, <...> );
```

### *Acompanhamentos Musicais*

A gramática de expressões ou acompanhamentos musicais 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*[19] para denotar 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, em que cada nota avança o tempo pelo valor da sua duração) ou em **paralelo** (separados por uma barra vertical |, criando uma bifurcação do contexto em dois, que irão correr em paralelo). 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:

**VELOCIDADE** 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 possível 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
```

### Teclados Virtuais

Para além de permitir declarar acompanhamentos musicais para serem tocados automaticamente, a linguagem permite declarar teclados virtuais. Associados às teclas do teclado podem estar notas, acordes, melodias, ou qualquer outro tipo de expressão suportado pela linguagem.

```
# Now an example of the keyboard. We can declare variables to hold state
$octave = 0;

$keyboard = @keyboard hold extend (v120) {
    # Keys can be declared to play notes
    a: C; s: D; d: E; f: F; g: G; h: A; j: B;
    # Or chords
    1: ^Cm; 2: BM; 3: AM;

    # Or any other expression, including code blocks that change the state
    z: { $octave = $octave - 1; };
    x: { $octave = $octave + 1; };
}

# The set_transform function allows changing all events before they are emitted by this keyboard
$keyboard::set_transform( $events => transpose( $events, octave = $octave ) );
```

Listing 3.2: Exemplo da sintaxe de teclados virtuais

Cada teclado aceita opcionalmente uma lista de modificadores, tais como:

**HOLD** Começa a tocar quando a tecla é premida e acaba de tocar quando é solta

**TOGGLE** Começa a tocar quando a tecla é premida, e acaba de tocar quando ela é premida novamente

**REPEAT** Quando a nota/acorde/música fornecida acaba de tocar, repete-a indefinidamente

**EXTEND** Em vez de tocar as notas de acordo com a sua duração, estende-as todas até a tecla ser levantada/premida novamente (quando usado em conjunto com **hold** ou **toggle**, respetivamente)

Também é possível passar uma expressão opcional entre parênteses que irá ser aplicada a todas as notas do teclado (no exemplo acima especificando um instrumento e o volume das notas com (:violin v120)), evitando assim ter de a repetir em vários sítios.

Para além disso, será possível controlar muitos mais aspectos do teclado atribuindo-o a uma variável e chamando funções a partir daí, como por exemplo efetuar transformações nos eventos e notas emitidos, ou simular o carregar e soltar das teclas.

```
# The set_transform function allows changing all events before they are emitted by this keyboard
$keyboard::set_transform( $events => transpose( $events, octave = $octave ) );
# It is possible to synchronize the keyboard with a grid to align the timings of key presses
# and releases to said grid
$keyboard::set_grid( Grid::new( 1, 4 ) );
# We can also control the keys manually
$keyboard::start( "ctrl+z" );
$keyboard::stop_all();
```

Listing 3.3: Exemplo da sintaxe proposta da linguagem

### 3.1.2 Arquitetura do Sistema

O sistema irá ser composto por um interpretador de linguagem desenvolvido em Python, acompanhado por uma interface de linha de comandos que faça uso do interpretador.

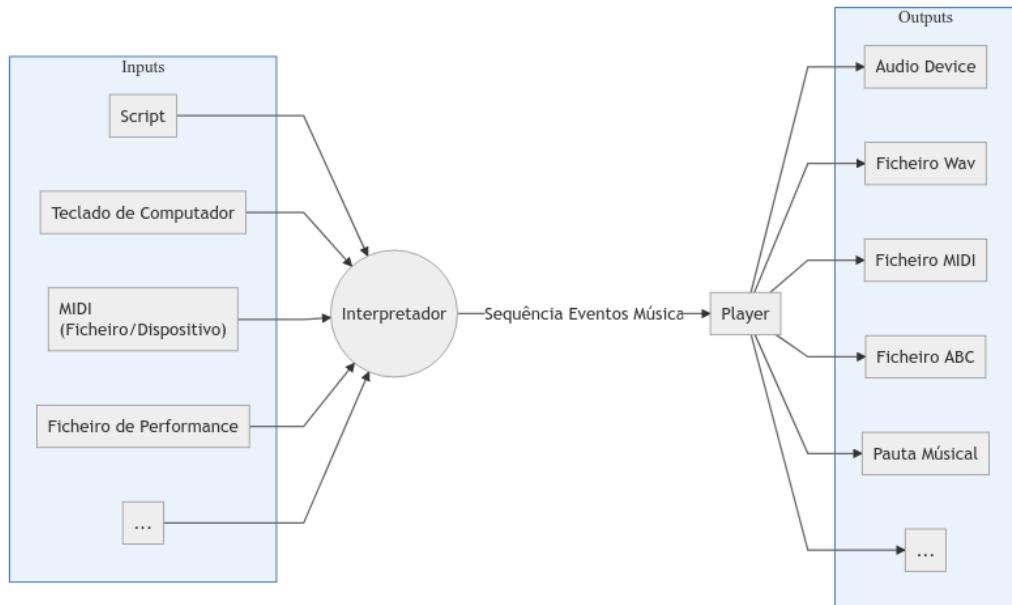


Figure 1: Arquitectura Geral do Projeto

O interpretador receberá um *script* obrigatório como input. Esse input poderá depois determinar quais os *inputs* (opcionais) que irá usar, como o teclado do computador, ficheiros ou teclados **MIDI**, ficheiros de performance (gravações de reproduções anteriores consistindo nos eventos em que as teclas foram premidas).

Os eventos gerados pelo *script* e os restantes *inputs* serão depois redirecionados para os diversos *outputs*, que podem ser as colunas do dispositivo, ficheiros **WAV** ou **MIDI**, ficheiros em formato PDF com a pauta musical, entre outros.

Toda a linguagem irá ser desenvolvida com extensibilidade em mente nos seguintes pontos:

**INPUTS** Permitir a criação de novos *inputs*, como estar à espera de mensagens por um *socket* ou de outros processos do computador.

**EVENTOS** A linguagem já disponibiliza uma variedade de eventos multimédia (como reproduzir notas, sons genéricos, ou mensagens para controlar dispositivos [MIDI](#)). Mas o objetivo é que criar e emitir eventos personalizados seja extremamente simples (e feito com uma só linha de código por evento, no mínimo). Obviamente, nem todos os eventos são suportados por todos os outputs (por exemplo, um ficheiro ABC não pode reproduzir um ficheiro [WAV](#)), mas neste caso os diversos *outputs* irão simplesmente ignorar os eventos que não estejam preparados para lidar.

**OUTPUTS** Para além dos *outputs* embutidos, permitir criar novos, tanto para outros formatos musicais, mas também para outros fins como controlar luzes, vídeo e imagens através da linguagem (em conjunto com os eventos personalizados) e permitir assim executar espectáculos multimédia completos a partir da linguagem.

**BIBLIOTECAS** Expôr funções, variáveis e objetos adicionais, bem como possivelmente acrescentar dinamicamente sintaxe à linguagem (através de macros).

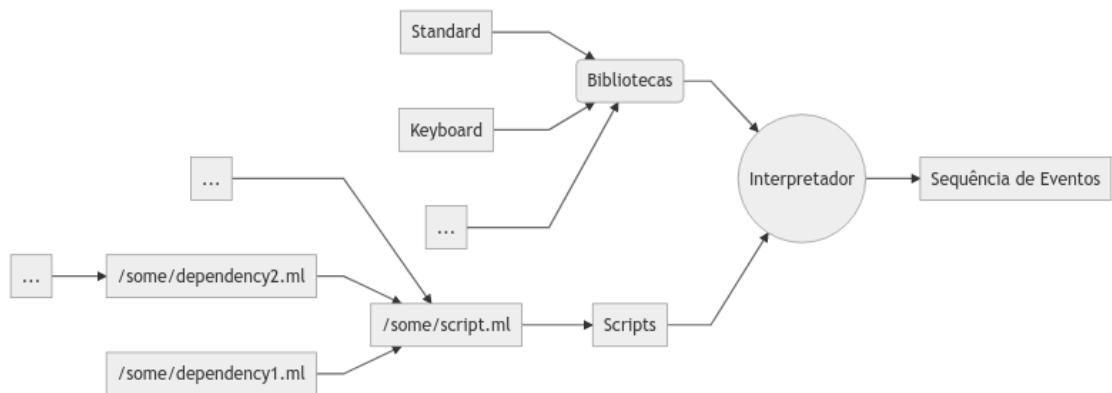


Figure 2: Arquitectura do Interpretador

O facto de a linguagem ser desenvolvida em *Python* significa que é extremamente prático extender a mesma devido a:

- Ubiquidade da linguagem Python tanto para programadores experientes como para iniciantes;

- Não ser necessário compilar o código para o executar;
- Sintaxe e expressividade da linguagem (apesar de vezes isso incorrer num custo de performance);
- Existência de *bindings* para bibliotecas desenvolvidas em C/C++ (como *numpy*), optimizadas para inúmeras tarefas que necessitem de maior *performance*/menor latência, que mesmo assim expõem uma interface agradável de usar em *Python*;

# 4

---

## MUSIKLA: CASOS DE ESTUDO

---

Neste capítulo iremos analisar possibilidades de uso da linguagem. Alguns desses exemplos são até já parcialmente ou totalmente funcionais quando executados no protótipo desenvolvido nesta fase inicial. Outros exemplos fazem uso de funcionalidades planeadas mas ainda não implementadas, e que serão devidamente identificados quando necessário. Nestes exemplos podem também ser usados pequenos excertos de músicas para demonstrar a utilização da linguagem, e a forma como esses excertos podiam ser representados com a nossa sintaxe.

### 4.1 TOCAR MÚSICA

```
# Title: Westworld Main Theme

:piano = (1; S6/8 T70 L/8 V120 );
:violin = :piano(41);

$chorus = :piano (A*11 G F*12 | A,6 A,5 G, F,6*2)*3;

$melody = :piano (r24 (:violin a3 c'3 d'3 e'9) r9 e'3 d'3 c'3 a9);

play( $chorus | $melody );
```

Listing 4.1: Exemplo da sintaxe para criação de música

#### Clicar para Reproduzir Áudio

Nas duas primeiras linhas deste exemplo podemos verificar a utilização de duas vozes (`:piano` e `:violin`). O piano ocupa a posição 1 da *soundfont* utilizada, enquanto que o violino utiliza a posição 41. Ao declarar o piano, podemos também definir um conjunto de configurações adicionais (como o compasso, a duração base das notas, e o volume com que são tocadas). Ao declarar o violino, podemos herdar as configurações de outro instrumento (neste caso o piano) e mudar apenas o necessário (a posição do instrumento).

Depois podemos ver a utilização de variáveis (`$chorus` e `melody`) para estruturar e guardar conjuntos de notas, neste caso. É possível ver também o quão conciso fica descrever padrões ou conjuntos de notas repetidas através do operador de repetição (\*). O operador de paralelo (|) permite depois tocar notas em paralelo ao mesmo tempo.

#### 4.2 DEFINIR UM TECLADO

No início do capítulo 3.1.1 podemos ver um pequeno excerto de música que é tocada autonomamente pela linguagem. Aqui poderemos ver como construir um teclado virtual personalizado para tocar essa música, bem como a sequência de teclas a premir para a reproduzir.

```
# Title: Soft to Be Strong
# Artist: Marina
V70 L1 T120;

fun toggle_sustain ( ref $enabled ) {
    if $enabled { cc( 64; 0 ) } else { cc( 64; 127 ) };

    $enabled = not $enabled;
};

$sustained = true;

@keyboard hold extend {
    a: ^Cm;    s: BM;    d: AM;    f: EM;    g: ^Fm;
};

@keyboard hold extend (V120) {
    1: ^c;    2: ^d;    3: e;    4: ^f;    5: ^g;
    6: b;    7: ^c';    8: ^d';    9: e';

    c: toggle_sustain( $sustained );
};
```

Listing 4.2: Exemplo da sintaxe para criação de teclados

#### Clicar para Reproduzir Áudio

No início deste exemplo podemos ver a declaração de uma função `toggle_sustain`, que recebe como parâmetro uma variável por referência. O que isto significa é que qualquer alteração ao valor da variável dentro desta função, reflete-se também na variável que for passada à função quando esta é chamada.

Lá dentro fazemos uso da função `cc` que permite controlar diversos controlos **MIDI**, neste caso, o controlo `64` refere-se ao pedal de *sustain* de um piano (que deixa as notas a tocar durante mais algum tempo mesmo depois da sua tecla ser levantada). O valor `0` (*zero*) que lhe é passado significa desligar esse pedal, e o valor `127` significa ligar esse pedal. No futuro, apesar de ser sempre possível recorrer a este tipo de funções de baixo nível, irão ser adicionadas à biblioteca *standard* as funções mais comuns (como por exemplo, `sustainoff()` e `sustainon()`).

Depois podemos ver a declaração de dois teclados virtuais (a linguagem permite mais do que um teclado ativo ao mesmo tempo). O primeiro mapeia a algumas teclas (`a`, `s`, `d`, `f` e `g`) o conjunto de acordes usados nesta música. Para além disso também define alguns modificadores a serem usados por este teclado (cujo significado é discutido no capítulo 3.1.1).

O segundo teclado funciona de forma similar, atribuindo às teclas de `1` a `9` notas individuais a serem tocadas. Neste teclado podemos também ver que notas musicais não são os únicos elementos que podem ser associados a teclas. Também é possível descrever expressões arbitrárias (como neste caso, a chamada da função `toggle_sustain( $sustained )` associada à tecla `c`).

Outro ponto a notar sobre o segundo teclado é a declaração entre parênteses (`V120`) que permite modificar o volume das notas tocadas por este teclado (que se sobrepõe ao volume global `V70` indicado no início do código). Isto é uma forma simples de prefixar configurações a todas as notas do teclado, evitando ter de copiar essas configurações para todas as notas.

#### 4.3 QWERTY KEYBOARD

A função `qwertyboard` é uma função planeada a ser incluída na biblioteca *standard* da linguagem. Aqui temos um exemplo simplificado do que essa função irá ser. Neste exemplo podemos observar funcionalidades mais genéricas da linguagem. Muitas dessas funcionalidades (arrays, métodos de objetos, funções anónimas) ainda não se encontram implementadas na versão atual do protótipo, mas servem como exemplo para o que a linguagem irá no fim permitir.

```
fun qwertyboard () {
    # Maps the lines on a keyboard to semitone offsets to List[ List[ str ] ]
    $lines = @[
        "qwertyuiop"::split(),
        "asdfghjkl"::split(),
        "zxcvbnm,.":split()
    ];

    $octave = 0;
    $semitone = 0;
```

```

$keyboard = @keyboard hold extend {
    [ $c for $c, $i in $lines::[ 0 ] ]: transpose( c; $i );
    [ $c for $c, $i in $lines::[ 1 ] ]: transpose( C; $i );
    [ $c for $c, $i in $lines::[ 2 ] ]: transpose( C,; $i );

    up => { $octave = $octave + 1 };
    down => { $octave = $octave - 1 };
    right => { $semitone = $semitone + 1 };
    left => { $semitone = $semitone - 1 };
};

$keyboard::set_transform( $events => $events + interval(
    semitone = $semitone,
    octave = $octave
) );

return $keyboard;
}

```

Listing 4.3: Exemplo da sintaxe proposta da linguagem

A primeira parte da função declara um *array* com as três linhas de caracteres presentes num teclado *QWERTY*. Isto permite-nos ao declarar as teclas no *keyboard*, fazê-lo de forma dinâmica (ao invés de associar a cada tecla uma nota manualmente).

Essa declaração, inspirada nas listas por compreensão do *Python*, funciona através de uma construção similar a um ciclo *for*. A variável *\$c* corresponde a cada item do *array* (neste caso, casa tecla), a variável *\$i* (opcional) permite-nos obter o índice da letra atual no *array*. A cada letra é associada a nota *C* transposta pelo índice da tecla.

Para além disso também podemos ver a declaração de mais quatro teclas (correspondentes às quatro setas do teclado) que permitem deslocar as notas tocadas por oitavas completas ou por semitonos. Para isso, estas teclas têm associadas uma expressão de bloco (identificada pelas chaves { e }). Lá dentro é possível meter uma instrução (ou opcionalmente várias, separadas por pontos e vírgulas ;). O valor da última expressão é o valor de retorno da expressão de bloco toda, pelo que é possível que uma tecla faça mais que uma coisa (altere o estado e no fim retorne ainda notas para serem tocadas, por exemplo).

Finalmente vemos também um exemplo do método *set\_transform*, um dos vários métodos que o objeto *Keyboard* irá ter e que permitem modificar ainda mais o comportamento dos teclados, desde transformar as teclas, definir grelhas de alinhamento, gravar as teclas premidas ou reproduzir uma dessas gravações, entre muitos outros.

Este método, que aceita uma função como argumento, permite transformar cada evento musical que o teclado emita. Neste caso, estamos a usá-lo em conjunto com as variáveis de

estado que declaramos anteriormente, para a cada evento, somar-lhe um intervalo composto pelos semitonos e oitavas definidos.

# 5

---

## PLANEAMENTO

---



Figure 3: Calenderização Prevista

O planeamento do projeto irá ser dividido em três fases principais, mesmo que na realidade estas fases acabem por se sobrepor um pouco. Paralelamente irá também ser decorrer a escrita da dissertação final que irá relatar o desenvolvimento do projeto.

**1<sup>a</sup> FASE Levantamento do Estado da Arte** Nesta fase irão ser investigadas as soluções já existentes, as ferramentas e bibliotecas que poderão ser usadas, bem como a teoria e os conceitos necessários para o desenvolvimento do projeto.

**2<sup>a</sup> FASE Arquitetura e Prototipagem** Previamente a desenvolver o produto final, poderá ser desenvolvido um ou mais protótipos a explorar diferentes vertentes do projeto, com o fim de avaliar quais os mais promissores e identificar com antecedência quais os problemas que podem surgir no desenvolvimento final.

**3<sup>a</sup> FASE Desenvolvimento Final** Combinar o conhecimento e as soluções desenvolvidas durante a fase de prototipagem num produto acabado.

**4<sup>a</sup> FASE Escrita da Dissertação** Relatar todos os aspetos relevantes da investigação e do desenvolvimento do projeto.

# 6

---

## CONCLUSÃO

---

O desenvolvimento do projeto envolve vários aspetos, desde o desenho da sintaxe e da gramática correspondente, até à geração de sons a serem guardados em ficheiros ou reproduzidos imediatamente em dispositivos áudio. Também engloba tanto aspetos mais genéricos da programação, como qual a melhor forma de implementar de forma opaca funções geradoras e funções assíncronas.

Mas para além disso também abrange quais as ferramentas e as metodologias são mais adequadas para a utilização da linguagem. É necessário para isso estudar com exemplos reais, qual a melhor forma de produzir e desenvolver música em formato textual.

No entanto, sendo a criação musical uma área tão ampla, é ínutil tentar sequer conseguir desenvolver uma ferramenta que cubra todos os cantos e sirva todas as necessidades dos seus utilizadores. É por isso que é importante apoiar o desenvolvimento do projeto nas vantagens que a linguagem *Python* fornece, quer a nível da sua facilidade de uso, popularidade, e fácil extensão sem necessidade de processos complicados de compilação.

O projeto deve ser por isso desenvolvido com extensibilidade em mente, tendo como objetivo principal servir como uma fundação estável capaz de ligar as diversas ferramentas existentes, não só na área da música, mas permitir ligar a música a outras áreas.

---

## BIBLIOGRAPHY

---

- [1] alda. <https://alda.io/>.
- [2] Abc notation. <http://abcnotation.com/>.
- [3] Abc notation standard v2.1, 2011.
- [4] Abc notation examples, 2011.
- [5] Faust. <https://faust.grame.fr/>.
- [6] Faust targets. <https://faust.grame.fr/doc/manual/#a-quick-tour-of-the-faust-targets>.
- [7] Faust libraries. <https://faust.grame.fr/doc/libraries/>.
- [8] Faust examples. <https://faust.grame.fr/doc/libraries/>.
- [9] Faust quick start. <https://faust.grame.fr/doc/examples/index.html>.
- [10] Sonic pi. <https://sonic-pi.net/>.
- [11] Sonic pi fx cheatsheet. <https://github.com/samaaron/sonic-pi/blob/master/etc/doc/cheatsheets/fx.md>.
- [12] Dangling else. [https://en.wikipedia.org/wiki/Dangling\\_else](https://en.wikipedia.org/wiki/Dangling_else).
- [13] Soundfont technical specification. Technical report, February 2006.
- [14] Timidity++. <http://timidity.sourceforge.net/>.
- [15] Wildmidi. <https://www.mindwerks.net/projects/wildmidi/>.
- [16] Fluidsynth. <http://www.fluidsynth.org/>.
- [17] Nathan Whitehead. pyfluidsynth. <https://github.com/nwhitehead/pyfluidsynth>, 2019.
- [18] Fluidsynth settings. <http://www.fluidsynth.org/api/fluidsettings.xml>.
- [19] Helmholtz pitch notation. [https://en.wikipedia.org/wiki/Helmholtz\\_pitch\\_notation](https://en.wikipedia.org/wiki/Helmholtz_pitch_notation).

# A

---

## APPENDIX

---

### A.1 GRAMÁTICA

```
main <- body EOF;

body <- statement ( ";" statement )* _ ";"? _
/ ""
;

// Statements
statement <- _ ( var_declaracion / voice_declaracion / function_declaracion / for_loop_statement
/ while_loop_statement / if_statement / expression ) _;

var_declaracion <- "$" namespaced _ "=" _ expression;

voice_declaracion <- ":" identifier _ "=" _ voice_declaracion_body;

voice_declaracion_body <- integer
/ "(" _ function_parameters _ ")"
/ ":" identifier _ "(" _ function_parameters _ ")"
;

function_declaracion <- "fun" _ namespaced _ "(" _ arguments? _ ")" _ "{" _ body _ "}";

arguments <- single_argument ( _ ";" _ single_argument )*;

single_argument <- single_argument_expr / single_argument_ref / single_argument_eval;

single_argument_expr <- "expr" _ "$" identifier;

single_argument_ref <- "ref" _ "$" identifier;

single_argument_eval <- "$" identifier;

for_loop_statement <- "for" _ "$" namespaced _ "in" _ value_expression _ ".." _ value_expression
_ "{" _ body? _ "}" ;
```

```

while_loop_statement <- "while" _ expression _ "{" _ body _ "}";

if_statement <- "if" _ expression _ "{" _ body _ "}" _ "else" _ "{" _ body _ "}";
      / "if" _ expression _ "{" _ body _ "}";

// BEGIN Keyboard
keyboard_declaraction <- "@keyboard" (_ alphanumeric)* ( _ group )? _ "{" ( _ keyboard_shortcut _ ;
;" )* _ "}";

keyboard_shortcut <- alphanumeric (_ "+" _ alphanumeric)* (_ alphanumeric)* _ ":" _ expression
      / string_value (_ alphanumeric)* _ ":" _ expression
      / "[" _ list_comprehension _ "]" (alphanumeric _ )* _ ":" _ expression
      / "[" _ value_expression _ "]" (alphanumeric _ )* _ ":" _ expression
      ;

list_comprehension <- expression "for" _ "$" _ namespaced _ "in" _ value_expression _ ".." _ value_expression
      / expression "for" _ "$" _ namespaced _ "in" _ value_expression _ ".." _ value_expression _ "if" value_expression
      ;
// END Keyboard

expression <- e music_expression;

music_expression <- sequence ( _ "|" _ sequence )*;

sequence <- value_expression ( _ value_expression)*;

group <- "(" _ expression _ ")";

block <- "{" _ body _ "}";

note <- note_accidental _ note_pitch ( _ chord_suffix )? ( _ note_value )?;

chord_suffix <- 'M' / 'm';

variable <- "$" namespaced;

function <- namespaced "(" _ function_parameters? _ ")";

function_parameters <- expression ( _ ";" _ expression )*;

chord <- "[" _ note ( _ note )* _ "]";

rest <- "r" ( _ note_value )?;

```

```

note_value
<- "/" _ integer
/ integer _ "/" _ integer
/ integer
;

note_accidental <- "^" / "^^" / "___" / "___" / "";

note_pitch
<- r"[cdefgab]" "'''*
/ r"[CDEFGAB]" ",,"*
;

modifier
<- r"[tT]" _ integer
/ r"[vV]" _ integer
/ r"[ll]" _ note_value
/ r"[sS]" _ integer _ "/" _ integer
/ r"[sS]" _ integer
/ r"[oO]" _ integer
;

instrument_modifier <- ":" ( identifier / "?" ) _ sequence;

value_expression <- expression expr_binary_op expression
/ expr_unary_op expression
/ string_value
/ number_value
/ bool_value
/ none_value
/ variable
/ function
/ keyboard_declaration
/ group
/ block
/ chord
/ note
/ rest
/ modifier
/ instrument_modifier
;

exp_binary_op <- "and" / "or" / ">=" / ">" / "==" / "!=" / "<=" / "<" / r"[+\\-*/]";

expr_unary_op <- "not";

```

```
string_value <- double_string / single_string;

double_string <- "\"" double_string_char* "\"";

double_string_char
<- "\\\\""
/ "\\\\\\"
/ r"[^\\]""
;

single_string <- ' single_string_char* "'';

single_string_char
<- "\\\\'"
/ "\\\\\\\'"
/ r"[^\']""
;

number_value <- float / integer;

bool_value <- "true" / "false";

none_value <- "none";

float <- r"[0-9]+.[0-9]+";

integer <- r"[0-9]+";

namespaced <- ( identifier "\\")* identifier;

identifier <- r"[a-zA-Z][a-zA-Z0-9\\_]*";

alphanumeric <- r"[a-zA-Z0-9\\_]*";

_ <- r"\t\r\n)*";

e <- "";

comment <- _ r"#[^\\n]*";
```

