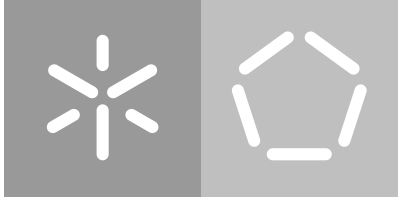
Universidade do Minho
Escola de Engenharia

Pedro M. Silva

Musikla
Music and Keyboard Language

June, 2020



Universidade do Minho
Escola de Engenharia

Pedro M. Silva

Musikla
Music and Keyboard Language

Master's Dissertation
Master's in Informatics Engineering

Work supervised by
José João Almeida

June, 2020

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

Lorem ipsum.

Acknowledgements

The acknowledgements. You are free to write this section at your own will. However, usually it starts with the institutional acknowledgements (adviser, institution, grants, workmates, ...) and then comes the personal acknowledgements (friends, family, ...).

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.

Keywords: Keywords (in English) ...

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 manipualação de aúdio digital, e envolvê-las numa linguagem de domínio específico que permita facilmente descrever composições músicais, gerar sons dinâmicamente, estudar propriedades da própria teoria músical, ou até criar teclados virtuais que toquem sons ou executem ações e podem ser utilizados para realizar espetáculos multimédia ao vivo.

Palavras-chave: Palavras-chave (em Português) ...

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Glossary

aliquam	tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.
computer	An electronic device which is capable of receiving information (data) in a particular form and of performing a sequence of operations in accordance with a predetermined but variable set of procedural instructions (program) to produce a result in the form of information or signals.
cras viverra	metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat.
donec nonummy	pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo.
integer sapien	est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus.
lorem ipsum	dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris.
maecenas lacinia	nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem.
morbi ac	orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.
morbi dolor	nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

nam lacus	libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi.
nam dui	ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo.
name arcu	libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo.
nulla malesuada	porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis.
sed lacinia	nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus.

Acronyms

AOT	Ahead Of Time
API	Aplication Public Interface
AST	Abstract Syntax Tree
CFG	Context Free Grammar
DAW	Digital Audio Workstation
DSL	Domain Specific Language
I/O	Input/Output
JIT	Just In Time
MIDI	Music Instrument Digital Interface
PEG	Parsing Expression Grammar
WAV	Waveform Audio File Format

Chapter 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 músicais, 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 intuítivo 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.

Chapter 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 músicais, com recurso a algum sintetizador externo, para gerar os sons, enquanto outras funcionam com base na manipulação direta de ondes 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** [**alda**] 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.

2.1.1.1 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.

```
1 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** [**abc-notation**] 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 músicais, ou mesmo ficheiros WAV (gerados através do fornecimento de um ficheiro SoundFont, por exemplo).

A complexidade da notação traz tanto vantagens como desvantages, 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 [**abc-notation-standard**] 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.

2.1.2.1 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 possíves, 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 maíuscula (**C**). As oitavas anteriores são identificadas por uma (ou mais) vírgula a seguir à nota com letra maíuscula (**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 [**abc-notation-exam**

2.1.3 Faust

A linguagem **Faust** [**faust**] 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 músicais. 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 [**faust-targets**].

A linguagem vem embutida com uma biblioteca extremamente completa [**faust-libraries**] 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).

2.1.3.1 Exemplos

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

```
1 import("stdfaust.lib");
2 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.

```
1 import("stdfaust.lib");
2 ctFreq = 500;
3 q = 5;
4 gain = 1;
5 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).

```
1 import("stdfaust.lib");
2 ctFreq = hslider("[0]cutoffFrequency",500,50,10000,0.01);
3 q = hslider("[1]q",5,1,30,0.1);
4 gain = hslider("[2]gain",1,0,1,0.01);
5 t = button("[3]gate");
6 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** [**sonic-pi**] 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, arpégios 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 simultâneamente. 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 [**sonic-pi-fx**]).

Para além das capacidades músicais, 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 tudas 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 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).

2.1.4.1 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 aleatóriamente 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 aleatóriamente uma dessas notas para tocar.

2.2 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 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 repercussões práticas entre as duas classes de gramáticas deve-se á semântica 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 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 ambíguidades, 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* [**dangling-else**] descrito nas duas classe de gramáticas:

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

Listing 2.9: Gramática

```
statement = ...
  | conditional_statement

```

```
4  conditional_statement = ...
5     | IF ( expression ) statement ELSE statement
6     | 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 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.

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 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.

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 atualemente na versão 2.04, lançada em 2005 [**soundfont**]. 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++ [**timidity**], WildMIDI [**wild-midi**] e FluidSynth [**fluidsynth**].

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 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).

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

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

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

# Aqui nao 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 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 [**soundfont**] (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** [**pyfluidsynth**], 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** [**fluidsynth_settings**] 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

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 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 ($0 + 4 * 12$). 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 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.

Chapter 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 indústriais 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 músical 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 díficil 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 **potêncialmente 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 músicais, 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 músicais, 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 construit 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 (atribuido a uma variável ou passado a uma função, por exemplo), esse valor músical 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-músicais, 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 .1. MAs antes de abordarmos em mais detalhe como irá der desenhada a gramática da linguagem, podemos aborar dois pequenos exemplos que demonstram a geração de notas músicais.

```
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 músicais (notas, acordes, etc).

Teclados Virtuais Açúcar sintático para facilitar a declaração de teclados virtuais. Para a lém de alguns construtores próprios, faz uso das expressões gerais e de acompanhamentos músicais descritas acima.

3.1.1.1 Instruções e Expressões

As instruções e expressões da gramática são baseadas nas linguagens como C e JavaScript. Chavetas são usadas para delenear 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>, <...> );`

3.1.1.2 Acompanhamentos Musicais

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*[**helmholtz-pitch-notation**] para denotar as respetivas oitavas. Podem também ser seguidas de um número ou de uma fração, indicando a duração da nota.

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

Listing 3.2: Exemplos de notas

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:

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
```

3.1.1.3 Teclados Virtuais

Para além de permitir declarar acompanhamentos músicais 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; };
}

```

```
15 # The set_transform function allows changing all events before they are emitted by this
       ↪ keyboard
16 $keyboard::set_transform( $events => transpose( $events, octave = $octave ) );
```

Listing 3.3: 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 novamete

repeat Quando a nota/acorde/música fornecida acaba de tocar, repete-a indefenidamente

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 aspetos 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.

```
1 # The set_transform function allows changing all events before they are emitted by this
      ↪ keyboard
2 $keyboard::set_transform( $events => transpose( $events, octave = $octave ) );
3 # It is possible to synchronize the keyboard with a grid to align the timings of key presses
4 # and releases to said grid
5 $keyboard::set_grid( Grid::new( 1, 4 ) );
6 # We can also control the keys manually
7 $keyboard::start( "ctrl+z" );
8 $keyboard::stop_all();
```

Listing 3.4: 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.

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).

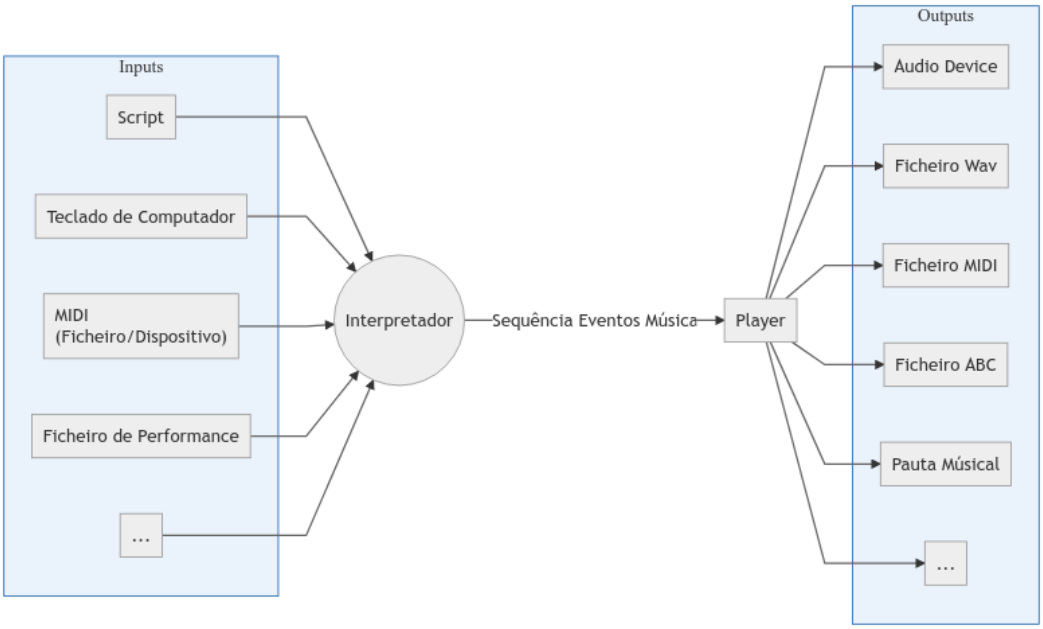


Figure 3.1: Arquitectura Geral do Projeto

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 customizados 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 músicais, 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).

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

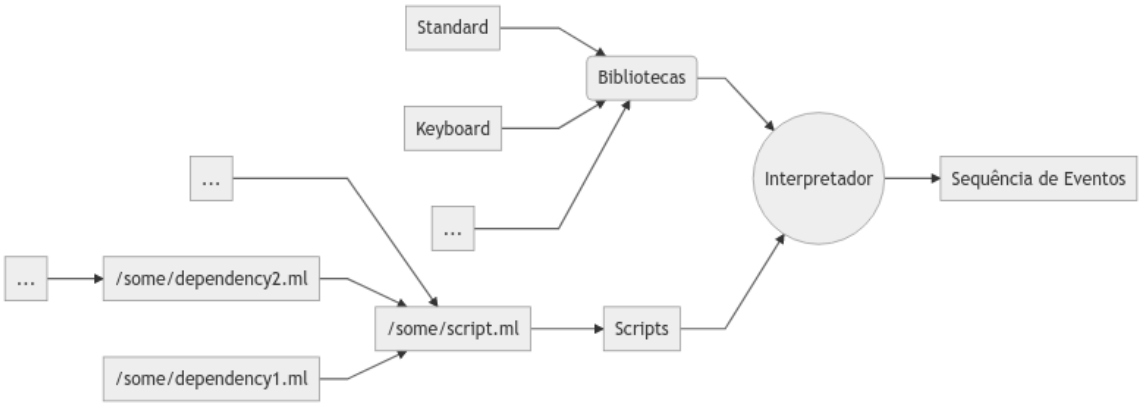


Figure 3.2: Arquitectura do Interpretador

- Ubiquidade da linguagem Python tanto para programadores experiêntes 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*;

Chapter 4

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

```
1  # Title: Westworld Main Theme
2
3  :piano = (1; S6/8 T70 L/8 V120 );
4  :violin = :piano(41);
5
6  $chorus = :piano (A*11 G F*12 | A,6 A,5 G, F,6*2)*3;
7
8  $melody = :piano (r24 (:violin a3 c'3 d'3 e'9) r9 e'3 d'3 c'3 a9);
9
10 play( $chorus | $melody );
```

Listing 4.1: Exemplo da sintaxe para criação de música

Nas duas primeiras linhas deste exemplo podemos verifica 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

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 modificares a serem usados por este teclado (cujo significado é discutido no capítulo 3.1.1.3).

O segundo teclado funciona de forma similar, atribuíndo às teclas de 1 a 9 notas individuais a serem tocadas. Neste teclado podemos também ver que notas músicais 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.

```
1  fun qwertyboard () {
2     # Maps the lines on a keyboard to semitone offsets to List[ List[ str ] ]
3     $lines = @[
4        "qwertyuiop"::split(),
5        "asdfghjkl"::split(),
6        "zxcvbnm,."::split()
7     ];
8
9     $octave = 0;
10    $semitone = 0;
11
12    $keyboard = @keyboard hold extend {
13       [ $c for $c, $i in $lines::[ 0 ] ]: transpose( c; $i );
14       [ $c for $c, $i in $lines::[ 1 ] ]: transpose( C; $i );
15       [ $c for $c, $i in $lines::[ 2 ] ]: transpose( C,; $i );
16
```

```
17        up => { $octave = $octave + 1 };
18        down => { $octave = $octave - 1 };
19        right => { $semitone = $semitone + 1 };
20        left => { $semitone = $semitone - 1 };
21    };
22
23    $keyboard::set_transform( $events => $events + interval(
24        semitone = $semitone,
25        octave = $octave
26    ) );
27
28    return $keyboard;
29 }
```

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 *QUERTY*. 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 desclocar as notas tocadas por oitavas completas ou por semitons. Para isso, estas teclas têm associadas uma expressão de bloco (identificada peloas chavetas `{` 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 músical 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 semitons e oitavas definidos.

Chapter 5

Desenvolvimento

O desenvolvimento do projeto pode ser dividido de grosso modo em três camadas. Nelas são cobertos um grupo abrangente de aspetos tanto da área do processamento de linguagens e do desenvolvimento de DSL's, da teoria musical, e do processamento digital de audio.

Na camada da linguagem esteve mais proeminente a área de processamento de linguagens, por motivos óbvios. Mas nas decisões tomadas durante o desenvolvimento desta camada, estiveram sempre presentes também as necessidades específicas que a teoria musical (e a sua notação) impõeem numa linguagem de programação.

Do mesmo modo, o interpretador faz claramente uso de tópicos do domínio do processamento de linguagens, mas é ainda mais fortemente influencidado pelas restrições e requisitos impostos pela componente musical da linguagem. Esta influencia a forma e a semântica da execução dos vários operadores disponibilizados.

A última camada, de desenvolvimento de uma biblioteca, composta pelos objetos e procedimentos que têm como objetivo facilitar a utilização da linguagem. Para isso foi necessário identificar os casos de utilização mais comuns e prioritários, de modo a guiar a construção destas interfaces para refletirem uma utilização real da aplicação.

5.1 Linguagem

A camada sintática da aplicação pode ser conceptualmente dividida em duas fases:

Sintaxe Esta fase caracteriza-se por delinear qual a sintaxe usada pela linguagem, bem como os construtores e operadores suportados;

Parser Nesta fase foi desenvolvido um *parser* em *Python*, responsável por converter o código fonte da linguagem numa Abstract Syntax Tree (AST);

No entanto, a realidade é que a abordagem seguida (não só nesta camada mas como em todo o projeto) foi mais iterativa, dividindo cada faze em porções semi-independentes e intercalando as várias porções das diversas fases. Esta abordagem tem a vantagem de permitir ir testando e experimentando com o projeto mais cedo do que seria possível com um modelo de desenvolvimento em cascata.

5.1.1 Sintaxe

A sintaxe da linguagem é bastante inspirada nas usualmente chamadas linguagens da família C, com recurso a parênteses curvos e chavetas para delínear os vários blocos da linguagem. No entanto, as expressões são complementadas com um novo conjunto de literais e operadores dedicados a componente músical da linguagem. Conseguir juntar estes dois mundos trás consigo alguns desafios que serão discutidos mais em detalhe em cada umas das secções seguintes.

5.1.1.1 Literais

Literais referem-se ao conceito de sintaxe desenhada com o propósito de descrever data (literal) no código. São usados em quase todas as linguagens de programação (e na nossa também) para descrever números, *strings* e booleanos.

A maior diferença nesta área entre a nossa linguagem foi a adição de literais responsáveis por modelar conceitos músicais, como notas, pausas e acordes. Esta sintaxe, tal como já foi mencionado anteriormente, foi inspirada pelo projecto *abc notation*, com algumas modificações.

5.1.1.2 Notas e Pausas

A sintaxe de notas descrição de notas é composta por quatro componentes: **acidentais**, ***pitch*** (obrigatório), **oitava** e **duração**.

```
[_^]*
[a-gA-G]
[',]*
([0-9]*\/)?[0-9]*
```

Listing 5.1: Expressão regular que identifica uma nota (quebras de linha apenas para claridade de leitura)

O *pitch* refere-se à nota (ou frequência) que deve ser tocada. O **C médio** (também conhecido como C4) é descrito simplesmente como `C`. É possível descer uma ou mais oitavas acrescentando uma ou mais vírgulas `,`. Para subir uma oitava, podemos primeiro substituir as letras maíusculas por minúsculas. Para subir mais oitavas, podemos acrescentar uma ou mais pelicas `'`. Para subir ou descer semitons, podemos preceder a notas com os acidentais `_` e .

As pausas são mais simples, sendo compostas simplesmente pela letra `r` seguida da sua **duração** (usando as mesma sintaxe das notas).

5.1.1.3 Acordes

Para definir acordes na linguagem, colocam-se várias notas dentro de parenteses retos. A notação usada para cada nota inclui os seus três primeiros componentes (acidentais, *pitch* e oitava), mas exclui a duração. Em vez de definir a duração em cada nota, esta é definida globalmente no acorde após fechar os parenteses retos.

Por conveniência, para evitar ao utilizador ter de introduzir todas as notas de um acorde manualmente, temos uma sintaxe abreviada para os tipos de acordes mais comuns, onde é apenas necessário introduzir a nota base seguido do tipo de acorde.

	Abreviações
Tríades	M, m, aug, dim, +
Quinta	5
Sétimas	m7, M7, dom7, 7, m7b5, dim7, mM7

Table 5.1: Lista de abreviaturas possíveis de serem acrescentadas a seguir a uma nota para especificar um acorde.

A decisão de envolver cada acorde com parênteses retos deveu-se ao facto de muitas abreviaturas serem já populares no domínio da notação musical, e como tal o ideal era não as mudar. No entanto, algumas dessas abreviaturas poderiam entrar em conflito com outros componentes da declaração da nota. Por exemplo, `C7` poderia ser tanto um acorde de sétima como uma nota com duração de 7. Com a separação por parenteses retos, a ambiguidade deixa de existir, sendo obvio que `[C7]` é um acorde de sétima, e `C7` é uma nota com duração 7.

```
1 [CFG]/4 [^Fm] [C5]2
```

Listing 5.2: Exemplos de três definições de acordes possíveis

5.1.1.4 Modificadores

Para além de permitir descrever notas, também é possível ter modificadores de contexto que permitem alterar certas propriedades das notas e acordes. Duas destas propriedades (duração e oitava) podem ser depois customizadas em cada nota ou acorde, como já vimos. No entanto, em vez de estes valores substituirem simplesmente os valores predefinidos, eles "complementam-se".

Ou seja, se declarar-mos por exemplo que a duração base das notas é $\frac{1}{4}$. Quando definirmos alguma nota a seguir com a duração de $\frac{1}{2}$, a sua duração real irá ser calculada da seguinte forma $\frac{1}{4} \times \frac{1}{2} = \frac{1}{8}$.

Do mesmo modo, quando definimos por exemplo a oitava base como 5 (o valor predefinido é 4), a nota `C,` passa a representar a oitava $5 - 1 = 4$ (por predefinição seria $4 - 1 = 3$).

Podemos então ver a lista dos modificadores aceites pela linguagem, bem como exemplos de utilização e os seus respetivos valores predefinidos (usados quando nenhum modificador é aplicado).

Nome	**Modificador**	**Exemplo**	**Predefinição**
Instrumento	IN	I46	I0
Velocidade	VN	V100	V127
Tempo	TN	T120	T60
Duração	LN L/N LD/N	L2 L/4 L3/8	L1
Oitava	ON	O2	O4
Compasso	SD/N	S3/4	S4/4

Table 5.2: Lista de modificadores e exemplos da sua utilização

5.1.1.5 Variáveis e Funções

Uma das consequências da introdução da sintaxe de notas literais foi a impossibilidade de ter variáveis com certos nomes. Uma variável chamada `a`, por exemplo, iria entrar em conflito com a nota do mesmo nome. Do mesmo modo, uma variável chamada `i1` iria entrar em conflito com o modificador de instrumento.

Em vez de criar casos de excepção para as variáveis que possam ter nomes que conflitam com outros construtores sintáticos, seguimos o exemplo de outras linguagems como *PHP*, *Perl* ou *Powershell*, e decidimos prefixar as nossas variáveis com o caracter `$`.

No caso das funções foi possível evitar a ambiguidade (e do mesmo modo a obrigatoriedade de as prefixar com um caracter) devido ao facto de as funções terem obriugatoriamente um par de parênteses (sem espaço entre o nome da função) quando são chamadas.

No entanto não quer dizer que as funções passaram imunes à introdução das literais de música. Uma vez que a vírgula é usada para mudar a oitava de uma nota (e foi escolhida de forma a manter compatibilidade com a sintaxe do projeto *abc notation*, existem casos em que esta não pode ser utilizada para separar os argumentos passados a uma função.

Por exemplo, dada a expressão `function_name(C, A, $a, 2)`, podemos concluir que a mesma tem quantos argumentos? A resposta correta seria dois, pois as duas primeiras vírgulas poderiam pertencer à nota ou ao separador da função. Mas neste caso a nota teria prioridade, pelo que `C, A, $a` seria o primeiro argumento, e `2` seria o segundo.

A solução tomada inicialmente foi utilizar ponto-e-vírgula para substituir a vírgula como separador de argumentos nas funções. E enquanto iso resolveu os problemas de ambiguidade, tornou-se óbvio à medida que que a linguagem foi avançando, que a prevalência da vírgula como separador em quase todas as linguagens de programação mais populares fazia com que houvesse um custo mental de mudança de contexto sempre que alguém mudava de alguma linguagem para a nossa, e vice-versa.

A solução escolhida no final foi um compromisso entre as duas opções: tanto a vírgula como o ponto-e-vírgula podem ser usados como separadores de argumentos, com a exceção de quando o argumento é uma nota musical, onde o ponto-e-vírgula tem de ser obrigatoriamente usado. Assim sendo, poderiamos

rescrever o exemplo anterior da seguinte forma `function_name(C; A; $a, 2)`, passando a função a receber agora os quatro argumentos como seria inicialmente esperado.

5.1.2 Parser

Para implementar o parser da linguagem, foi utilizado o módulo *Python* **Arpeggio**, um módulo que implementa um algoritmo de *parsing* descendente recursivo como recurso a *memoization* para melhora da performance. A gramática utilizada pode ser vista em maior detalhe no apêndice .1.

```
main <- body EOF;

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

// Statements
statement <- _ ( var_declaration / voice_declaration / function_declaration /
    ↪ for_loop_statement / while_loop_statement / if_statement / expression ) _;

var_declaration <- "$" namespaced _ "=" _ expression;
// ...
```

Listing 5.3: Excerto da gramática desenvolvida

A gramática PEG desenvolvida para o projeto foi depois complementada por uma classe **Parser**, responsável por gerar a AST da linguagem. Para isso recorremos ao *Visitor Pattern*, com um método para cada regra não-terminal da gramática (todos prefixados com `visit_`).

```
def visit_body ( self, node, children ): ...

def visit_comment ( self, node, children ): ...

def visit_statement ( self, node, children ): ...

def visit_var_declaration ( self, node, children ): ...
```

Listing 5.4: Métodos responsáveis por criarem a AST

5.2 Interpretador

As linguagens compiladas usualmente recorrem à compilação Ahead Of Time (AOT), em que o código é transformado em código máquina com antecedência (durante a fase de compilação). Esta é a solição que consegue geralmente oferecer melhor *performance*, menor consumo de memória e tempos de *startup* mais rápido. Por outro lado, obriga a um passo de compilação separado sempre que o código fonte é

alterado, e regra geral necessita de tipos de dados estáticos. No que toca a linguagens interpretadas temos mais opções. Podemos então dividir os seus modos de execução em três categorias distintas, cada uma com possíveis vantagens e desvantagens, bem como diferenças de dificuldade de implementação bastante salientes.

Interpretadores Também chamados por vezes como interpretadores *tree walk*, são usualmente os mais simples de implementar (mas também os mais lentos a executar). O seu conceito baseia-se no *design pattern* homónimo, em que as operações da linguagem são modeladas numa árvore, geralmente igual ou similar à estrutura da AST. Para executar uma expressão é chamado um método na raiz da árvore, e esse método irá chamar recursivamente os métodos das suas sub-expressões, passando o estado como argumentos da função, e recebendo o resultado pelo valor retornado da função.

Bytecode VM São chamadas máquinas virtuais (VM) porque o seu comportamento assemelha-se mais ao comportamento dos processadores reais dos computadores. As expressões da árvore de sintaxe abstrata são previamente convertidas numa sequência linear de instruções mais simples (geralmente compactadas em binário para melhor *performance*, também referido como *bytecode*. Cada instrução é depois executada dentro de um ciclo. Esta solução fornece um balanço entre facilidade de implementação e velocidade de execução (evitando a grande quantidade de chamadas recursivas de funções presentes nos interpretadores).

Just In Time O método mais complexo (mas também o que oferece melhor *performance* para tarefas pesadas). O código é executado inicialmente por um dos dois métodos anteriores, de modo a recolher estatísticas sobre qual o tipo de execução mais comum do código. Após esta recolha, é gerado código máquina optimizado para os tipos de dados mais comuns de uma variável, ou para os caminhos de execução mais prevalentes, para que seja possível da próxima vez que o mesmo pedaço de código seja executado, isso ocorra com recurso ao código máquina. Este processo é gerlamente repetido ao longo da execução do programa, sendo que caso a versão de código máquina gerada fique desatualizado (o tipo de dados usualmente passados a uma função mudem), essa porção de código seja invalidada e eventualmente substituida por uma nova versão mais adequada.

É possível ver que a solução ideal seria sempre a compilação Just In Time (JIT), que evita uma fase de compilação explícita e forçada ao utilizador, mas ao mesmo tempo consegue fornecer *performance* competitiva para tarefas mais exigentes. No entanto é inevitável concluir que esta solução impõe custos de desenvolvimento astronómicos, e implica equipas de grande dimensão e tempos de desenvolvimento extremamente longos.

Desta forma a nossa escolha reside logicamente entre a solução de **Interpretador** e um simples **Bytecode**. Acabamos por escolher a primeira opção pelas seguintes razões:

- A geração de eventos músicais é relativamente computacionalmente barata (mesmo admitindo algumas dezenas de eventos músicais por segundo).
- A componente mais pesada geralmente reside na sintetização dos eventos músicais (*note on, note off*) em *streams* de audio, mas esta tarefa é encaminhada para bibliotecas desenvolvidas em linguagens de baixo nível.
- A possibilidade de correr código *Python* no meio de qualquer parte da nossa linguagem já fornece um bom meio termo para quando é necessária mais alguma *performance* (sem sacrificar demasiado a simplicidade de uma linguagem destinada primariamente a músicos e não engenheiros informáticos).
- Também a facilidade de implementação de um interpretador significou uma velocidade de ordens de magnitude superior durante a implementação da linguagem e da prototipação de funcionalidades.
- Uma posterior modificação do interpretador para uma *bytecode VM* mais eficiente seria possível estando a linguagem mais estabilizada, e seria simples manter uma API virtualmente 100% compatível.

Em conclusão, cada operação foi implementada através de um método `eval()` em cada classe da AST, recebendo uma variável de contexto como *input*, e usando depois o valor retornado pela função como resultado da avaliação da expressão.

5.2.1 Contexto

A variável de contexto guarda o estado da execução, e deve conter toda a informação necessária para cada instrução poder executar. Os seus conteúdos podem ser divididos em três componentes:

Timestamp Esta propriedade é um simples inteiro que permite às expressões de geração de eventos músicais saberem qual o tempo virtual atual. Este tempo virtual é manipulado pelos vários operadores. O operador sequencial por exemplo, que recebe uma lista de expressões e emite uma lista de eventos músicais uns a seguir aos outros, avança este cursor para o fim do último evento antes de passar o contexto para avaliar a expressão seguinte.

Voz Objeto que contém as várias propriedades músicais que devem ser aplicadas durante a geração de eventos, tais como a duração base das notas, o compasso ou as batidas por minuto.

Símbolos Um contentor que permite aceder e modificar os símbolos disponíveis na linguagem (tais como variáveis e funções).

Os contextos foram desenhados com o intuito de serem leves, daí apenas conterem referências para três variáveis. Desta forma é possível criar cópias dos contextos e permitir que operações variadas estejam a executar ao mesmo tempo com diferentes contextos.

Para percebermos a razão desta necessidade, basta pensarmos no operador paralelo. Se colocarmos duas expressões músicais que devem tocar ao mesmo tempo, a geração de uma (ou mais) notas na primeira expressão não deve afetar o `timestamp` da segunda. Para isto, cada uma das expressões deve receber uma cópia do contexto (com o *timestamp* inicial igual). Quando as duas expressões terminarem, no entanto, é importante que o contexto original (que gerou as duas cópias) atualize os seu *timestamp* para qual for a expressão mais longa.

Se prestarmos atenção, podemos estar aqui a detetar um padrão bastante comum na área da programação: o modelo **fork-join**.

E é fácil perceber que faz sentido. Apesar de estarmos a lidar com notas e eventos músicais, fundamentalmente estamos a criar ramos paralelos de execução, e queremos no final aguardar o seu resutado e unificá-lo com o ramo original. Se substituirmos ramo por *thread* para o caso da programação, ou *contexto* para o nosso caso, vemos a equivalência de contextos.

Como tal, para além do estado (as três variáveis) que o objeto de contexto engloba, este providencia também algumas operações bastante simples e úteis:

Fork Cria uma cópia do contexto, podendo receber opcionalmente também um inteiro como argumento com vista a substituir o valor do *timestamp* do contexto pai. Como segundo argumento pode também receber um novo *scope* de símbolos, algo que iremos aprofundar mais no capítulo seguinte.

Join Recebe uma lista de contextos como argumento, e avança o *timestamp* para o maior encontrado nessa lista.

Seek A operação de mudar o *timestamp* do contexto atual.

Os contextos não têm (nem precisam) de referências para outros *contextos-pai* ou *contextos-filhos*, de modo a que não é preciso grandes preocupações relativamente a fugas de memória com a criação de novos contextos: apenas são mantidos em memória os contextos que têm referências para eles, e que portanto estão ainda em uso.

5.2.2 Scope de Símbolos

Cada contexto tem uma referência para o *scope* de símbolos a que tem acesso. Nestes scopes são guardadas as variáveis e funções que o utilizador (e as bibliotecas standard da linguagem) declararem. Mais uma vez, cada objeto de *scope* é composto por três propriedades: uma referência ao seu **antecessor** (ou pai), uma tabela de *hash* com os **símbolos**, e uma *flag* booleana para designar este *scope* como **opaco**.

O próprio conceito de *scope* implica por si só uma hierarquia, e de facto cada objecto *scope* guarda uma referência para o seu parente (ou para o valor nulo, caso seja o *scope* raiz). Esta propriedade é utilizada para as operações disponíveis, tanto para pesquisar, como para atribuir valores a símbolos do *scope* atual, algo que iremos verificar mais a seguir.

Pesquisa A operação de pesquisa por um símbolo começa por procurar o símbolo no próprio *scope*, e caso não encontre nada, navega recursivamente para o *scope* pai para efetuar a pesquisa.

Atribuição A operação de atribuição segue a mesma filosofia da operação de pesquisa, procurando o *scope* onde o símbolo a atribuir está guardado. No entanto, esta pesquisa decorre até encontrar o primeiro *scope* opaco. Um *scope* opaco não impede a leitura de valores de *scopes* superiores, mas impede a escrita. Por exemplo, uma atribuição dentro de uma função cria apenas uma variável dentro do *scope* dessa função.

Se encontrar o símbolo nalgum *scope*, então muda o seu valor nesse *scope* onde está declarado. Caso contrário, é criado um novo símbolo no *scope* original onde a operação de atribuição foi iniciada.

Enumeração A operação de enumeração permite listar quais os símbolos vísiveis a partir de um *scope*. Ela permite listar não só os símbolos do próprio *scope*, mas também dos seus pais, tendo o cuidado para não retornar símbolos que tenham sido obscurecidos por um *scope* mais em baixo. Esta é útil para importar símbolos do *scope* de um módulo para outro, ou para implementar funcionalidades como *autocomplete* sobre um script em execução (e permitir sugerir os símbolos disponíveis).

Apontadores Como vimos na atribuição, é ímpossível alterar valores de variáveis globais dentro de *scopes* opacos (como dentro de funções, por exemplo). Para colmatar isso, similar ao operador `global` e `nonlocal` do *Python*, é possível instruir um *scope* a criar um apontador para um símbolo declarado noutro *scope*, de forma a que quando ocorre alguma atribuição, a alteração é replicada no seu *scope* original.

5.2.3 Módulos

O interpretador dispõe também da possibilidade de executar código separado em vários ficheiros, cada um sendo considerado um módulo. O utilizador pode configurar uma lista de pastas (num conceito similar à variável de ambiente `$PATH`) onde serão procurados módulos quando alguma instrução `import` é avaliada. Também é possível passar um caminho absoluto ou relativo ao módulo que iniciou a importação.

Cada ficheiro importado é executado apenas uma vez durante a primeira importação, e no final o seu *scope* é guardado em *cache* para ser usado em futuras importações.

Cada instrução de importação é dividida em duas fases: primeiro, o caminho passado é resolvido no caminho real e absoluto do ficheiro. Só depois é verificada na cache se o ficheiro já foi importado antes. Uma vez obtido o seu *scope*, os seus símbolos são enumerados e copiados para o scope que efetuou a importação (com símbolos que comecem com um *underscore* sendo tratados como símbolos privados e ignorados).

Para isso, quando o interpretador é inicializado, é criado um *scope* raiz chamado **prelude**, e um *scope* filho. Cada módulo importado é executado num *scope* criado sempre a partir do *prelude*. Desta forma, os símbolos aí guardados estão sempre acessíveis em todos os módulos importados.

5.2.4 Operadores Músicais

5.3 Biblioteca Standard

5.3.1 Inputs & Outputs

Durante a execução, os eventos músicais são representados em memória por objetos *Python* que derivam da classe base **MusicEvent**, e estão agrupados em sequências pelos objetos que derivam da classe **Music**.

A forma mais óbvia de criar estes objetos é utilizando a syntaxe de literais de música inspirada pelo projeto *abc notation* que pode ser utilizada dentro da própria linguagem.

No entanto, não há nenhuma restrição que force este a ser o único método de introdução ou geração de eventos músicais. De facto, existem já vários formatos de descrição de música, e a capacidade de os poder ler e traduzir para objetos músicais dentro da nossa linguagem, indistinguíveis daqueles que são criados pela sintaxe de literais, traz ínumeras vantagens e abre a possibilidade de este projeto se tornar como um canivete suísso da notação músical, capaz de ler, transformar e depois também escrever em diversos formatos diferentes.

Para além de ser capaz de ler (**input**) eventos músicais e transforma-los nos objetos em memória que a nossa linguagem expecta, é também importante poder, depois de os criar e transformar ao nosso gosto, poder fazer algo de útil com eles. Ou seja, escrever (**output**) estes eventos músicais em diversos formatos.

O verdadeiro objetivo da linguagem é que seja extensível, de modo a que qualquer pessoa capaz de programar *Python* possa connectar um novo *input* ou *output* para as suas necessidades específicas, sem necessitar de ter conhecimentos intrínsecos do funcionamento de toda a linguagem. Nesta secção iremos descrever alguns dos modos de *input* e *output* incluídos de base com a linguagem e que servem de prova de conceito do que é possível fazer com ela.

Os *outputs* podem ser globais, ou seja, aplicarem-se aos eventos emitidos pela aplicação. Neste caso são geralmente passados como argumentos de linha de comandos, ou adicionados durante a execução através da função `$script::player::add_sequencer()`. Também é possível gravar apenas expressões de música específicas com a função `save()`.

5.3.1.1 FluidSynth

Esta biblioteca serve apenas como **output** para a linguagem, e permite sintetizar sons a partir de ficheiros *SoundFont*. Os sons gerados podem depois ser passados diretamente para as colunas do computador, ou guardados em disco num ficheiro de música.

Para implementar este *output*, utilizamos o projeto *pyfluidsynth*[**pyfluidsynth**] para interoperar o nosso código *Python* com a biblioteca *FluidSynth*, escrita em *C*. Como os *bindings* do projeto em *Python*

não cobriam toda a API da biblioteca que nós necessitavamos, optamos por realizar um *fork* dos *bindings*, o que nos permitiu depois efetuar as alterações necessárias à nossa medida.

O objeto principal disponibilizado pelo *FluidSynth* é o objeto `Synth`. Este disponibiliza inúmeros métodos (baseados no *standard* MIDI) para ativar e desativar uma nota, mudar o instrumento, definir o valor de um controlo, entre muitas outras. Os métodos chamados neste objetos são aplicados imediatamente, pelo que se tivermos uma lista de ações (eventos músicais) a tomar com *timestamps* variados, temos de tratar do seu correto agendamento.

Felizmente a biblioteca também providencia um objeto para isso: `Sequencer`. Para o efeito registamos um *callback* na nossa aplicação que recebe o evento e é responsável por o aplicar ao `Synth`. Depois, sempre que algum evento é gerado, chamamos a função `Sequencer.timer()` e passamos-lhe o *timestamp* em que deve ser executado o evento, bem como o evento em si e o *callback* a chamar quando o temporizador concluir.

A maioria dos eventos da nossa linguagem são inspirados pelo *standard* MIDI, tal como a é desenhada a API do sintetizador do *FluiSynth*, pelo que aplicação dos eventos passa geralmente apenas por chamar a função correta e fornecer-lhe os parâmetros carregados pelo evento. Há no entanto uma exceção à regra, algo que iremos aprofundar mais à frente, e que é a possibilidade de reproduzir, para além de notas músicais, mas ficheiros músicais também. Infelizmente o *FluidSynth* não suporta diretamente este tipo de utilização, mas com alguma criatividade foi possível implementá-la usando as ferramentas que nos eram disponibilizadas.

A sintetização de notas efetuada pelo *FluidSynth* recorre aos ficheiros *Soundfont*, que podem ser descritos de uma forma muito simplista como um dicionário que associa a cada nota de cada instrumento um *buffer* contendo o som a ser reproduzido, mais algumas configurações que permitem ajustar o som gerado a diversas situações (duração da nota, velocidade, entre outras).

Estas fontes de som podem ser carregadas para o *FluidSynth* e através do método `Synth.sfload()` e passando-lhe o caminho do ficheiro. Mas para além de fontes carregadas do disco, também é disponibilizada uma estrutura `RamSFont` que permite criar, representar e editar uma fonte de som completamente em memória, que pode ser depois associada ao sintetizador pelo método `Synth.add_sfont()`.

Sempre que algum evento músical traz consigo um ficheiro de música para ser reproduzido, nós associamos o conteúdo desse ficheiro a uma nota de um instrumento virtual da `RamSFont` que criamos. Quando o mesmo ficheiro é pedido para ser reproduzido várias vezes, a mesma nota do mesmo instrumento é utilizada, evitando estar sempre a ler os conteúdos do ficheiro. Este evento músical é depois convertido num evento de reprodução de notas, com a informação da nota e do instrumento a usar sendo preenchidas automaticamente com o nosso instrumento virtual.

Desta forma, podemos reproduzir ficheiros arbitrários de som em conjunto com as notas músicais de uma forma transparente para o utilizador.

No entanto, um obstáculo que encontramos durante a implementação desta funcionalidade foi o facto de os sons associados a notas com um valor inteiro inferior a 15 (cada instrumento pode ter 128 notas, entre 0-127) tinham o seu *pitch* distorcido, mesmo quando eram passadas as *flags* para que os sons

fossem reproduzidos sem ajustamento do *pitch*. Este comportamento no entanto desaparecia acima de notas com o valor 15, pelo que adotamos esse valor como a nossa base (o que desperdiça 15 lugares em cada instrumento que poderiam ser associados a sons). Como na *SoundFont* podemos utilizar também vários instrumentos, isso significava que ainda assim conseguíamos reproduzir $127 \times 113 = 14351$ sons distintos, o que é bastante mais do que suficiente para a maioria dos casos.

5.3.1.2 MIDI

Mais uma biblioteca que achamos importante incluir de base foi o suporte para leitura e escrita de dados MIDI. Isto tanto pelo facto de que o *standard* MIDI é um dos mais utilizados no mundo da música, mas também porque a nossa implementação da linguagem, particularmente relativa à representação dos eventos músicais em memória, foi fortemente inspirada por este formato. Dessa forma, implementar funções de transformação dos não exigiu demasiado esforço. Para o efeito, utilizamos o módulo *Python* `mido`, que permite ler e escrever eventos MIDI, tanto de portas como de ficheiros em disco.

A nível de leitura (**input**), o módulo disponibiliza uma função `readmidi()` que permite ler eventos músicais de um ficheiro ou porta MIDI, returnando um objeto do tipo **Music**. A função aceita os seguintes parâmetros (sendo os parâmetros de ficheiro ou de porta mutualmente exclusivos):

file *(Opcional)* Quando presente, lê os eventos músicais de um ficheiro MIDI.

port *(Opcional)* Permite ler os eventos de uma porta em vez de um ficheiro. Este parâmetro aceita vários valores.

- **True** Se receber este valor, tenta listar as portas MIDI disponíveis e escolher a mais indicada para ler os eventos.
- **List[String]** Uma lista de nomes de portas para escutar. Os eventos das portas referidas são combinados numa única sequência de eventos, como se viessem todos da mesma porta.
- **String** O nome de uma única porta para estar à escuta de eventos.

voices *(Opcional)* Uma lista de vozes para mapear a cada canal MIDI a que os eventos pertencem. Quando nenhuma voz é especificada, a voz atual presente no contexto é utilizada para todos os canais.

cutoff_sequence *(Opcional)* Uma sequência de notas ou eventos músicais que, quando recebidos pelas portas MIDI, são utilizados como sinal para terminar imediatamente a leitura e retornar a sequência de eventos já capturada. È ignorada quando a fonte de leitura é um ficheiro.

ignore_message_types *(Opcional)* Uma lista de *strings* com os nomes de mensagens MIDI que devem ser ignoradas.

É também possível escrever (**output**) para portas e ficheiros MIDI de forma bastante simples. Este *output* é ativado automaticamente quando se grava um ficheiro com a extensão MIDI, ou quando se prefixa o nome de uma porta com a *string* `midi://`.

Ao escrever para ficheiros ou portas MIDI, é possível filtrar apenas os eventos de certas vozes, e mapear também os canais atribuidos a cada voz. Isto é extremamente útil para permitir ligar a aplicação a programas DAW, e permitir receber os eventos MIDI em faixas separadas (para aplicar efeitos ou transformações específicas a cada faixa).

5.3.1.3 ABC

5.3.2 Ficheiros de Som

Como já foi mencionado no sub-capítulo sobre o *output* **FluidSynth**, uma das funcionalidades que queriamos implementar na linguagem era a possibilidade de reproduzir ficheiros de som arbitrários, e integra-los com o resto da geração de eventos músicais.

Isto abre a possibilidade de incluir nos projetos que usem a linguagem sons gerados por outros programas, e mais uma vez vai de encontro ao nosso objetivo central de extensibilidade da linguagem. Podemos pensar assim que mesmo que a aplicação não suporte todo o tipo de geração de sons que alguma pessoa possa precisar, é sempre possível usar a aplicação para a maioria dos casos mais comuns que são suportados, e gerar ficheiros com recurso a alguma aplicação externa que colmatem as nossas necessidades mais específicas. Para utilizar-mos esta funcionalidade, temos à nossa disposição o método `sample()`.

```
1 A B sample( "cihat.wav" );
```

Listing 5.5: Exemplo de reproduzir um ficheiro a seguir a duas notas

Para facilitar todo o processo, os ficheiros de som devem seguir todos as mesmas especificações. Por conveniência, adoptamos as configurações suportadas pela biblioteca *FluidSynth* para blocos de som nas suas *SoundFonts*.

Configuração	**Valor**
Formato	WAVE
Sample Rate	41.100hz
Canais	2
Bit depth	16bit
Compressão	*N/A*

Table 5.3: Formato nativo suportado pelo FluidSynth

Para facilitar a utilização, é possível ao utilizador carregar ficheiros de som em formatos diferentes desde que tenha a ferramenta FFMPEG instalada no sistema. Quando isso ocorre, a linguagem converte

o ficheiro musical *background*, aplicando as configurações necessárias, e guarda-o depois em memória. Isto é especialmente útil para experiências rápidas e com ficheiros pequenos (até alguns *megabytes*).

Esta funcionalidade traz bastante simplicidade à utilização da linguagem, mas impõe um custo durante a inicialização de todos os programas. Para permitir aos utilizadores determinarem se os seus ficheiros vão necessitar de conversão, sem terem de recorrer a ferramentas externas para efetuar a verificação, disponibilizamos a função `is_sample_optimized()`, em conjunto com a função `optimize_sample()` para converter o ficheiro e gravar o resultado em disco.

```
1 if ( not is_sample_optimized( "cihat.wav" ) ) {
2    # Converts the file and saves it to disc
3    optimize_sample( "cihat.wav", "cihat_opt.wav" );
4 };
```

Listing 5.6: Verificar se um ficheiro de audio está optimizado, e convertê-lo caso contrário

Desta forma a conversão ocorre apenas uma vez, e de seguida o utilizador pode utilizar o ficheiro convertido e não se preocupar com a perda de performance sempre que usa o som num *script*.

5.3.3 Teclados Músicais

Para além de permitir construir expressões que geram sequências de eventos músicais dinâmicas, algo que esteve desde início no topo da nossa lista de prioridades foi sem dúvida adicionar suporte para a criação de teclados interativos. Sendo esta linguagem desenvolvida em computadores, não seria errado pensar que por teclado nos referimos aos teclados físicos dos computadores. E esses fazem certamente parte, mas quando nos referimos a teclados músicais, referimo-nos à possibilidade de descrever na nossa linguagem mapeamentos entre *eventos* e expressões músicais ou ações a executar quando esses eventos acontecem.

```
1 @keyboard {
2    a: print( "Tecla a carregada" );
3    b: d;
4 };
```

Listing 5.7: Exemplo de declaração de duas teclas

Esses eventos podem então ser teclas de um teclado, mas também de um piano conectado ao computador, ou eventos do rato, ou de qualquer outro dispositivo de I/O que as pessoas queiram adaptar, bastando para isso herdar a classe`KeyboardEvent`.

5.3.3.1 Tipos de Eventos

Os tipos de evento mais comum são teclas de teclado e de piano. Por essa mesma razão implementamos açúcar sintático na definição desses eventos (que são implementados pelas classes `KeyStroke` e `PianoKey`, respectivamente).

```
@keyboard {
   ctrl+c: ^c;
   [16]: d;
   [c']: e;
};
```

Listing 5.8: Declaração de três eventos, o primeiro é uma combinação de teclas, o segundo referência o *virtual key code*, e o terceiro uma nota MIDI

A lista de tipos de eventos suportados de base pela linguagem são os seguintes:

KeyStroke Este tipo de eventos referem-se às teclas do teclado. Para além de permitirem descrever teclas singulares, também suportam os modificadores *ctrl*, *alt* e *shift*. O evento não carrega consigo nenhum parâmetro extra.

Parâmetros: *N/A*

PianoKey Sinalizam quando uma nota é tocada e recebida pela porta MIDI que a aplicação esteja à escuta. Trazem consigo um parâmetro que identifica a velocidade com que a nota foi premida.

Parâmetros: `$vel`

MouseClick Evento ocorre quando algum dos botões do rato é premido. As informações sobre a posição em que o rato se encontra, bem como qual o botão premido e se foi premido ou levantado, são incluidas como parâmetros deste evento.

Parâmetros: `$x`, `$y`, `$button` `$pressed`

MouseMove Evento ocorre sempre que o rato é movido. Pode ser útil paar controlar alguma variável como se fosse um *slider*. Trás como parâmetros as coordenadas do rato.

Parâmetros: `$x`, `$y`

MouseScroll Evento despoletado quando a roda do rato é acionada. Para além de trazer informações sobre a posição do rato, indica também qual o valor que a roda se moveu, tanto na vertical (mais comum) como também na horizontal.

Parâmetros: `$x`, `$y`, `$dx`, `$dy`

Os **parâmetros** são algo que, como pudemos ver, é disponibilizado por quase todos os tipos de eventos. Eles dão-nos a possibilidade de saber que os eventos foram despoletados, mas saber propriedades sobre o que os despoletou. Estes parâmetros podem ser acedidos passando as variáveis desejadas com o nome do parâmetro (a ordem é irrelevante) à frente da declaração do evento. Essas variáveis podem depois ser acedidas dentro da ação do evento.

```
@keyboard {
   [keyboard\MouseMove] ($x, $y): print( $x, $y );
};
```

Listing 5.9: Teclado que imprime as coordenadas do rato sempre que ele se move

Para criar novos tipos de eventos, basta criar uma nova classe em *Python* que derive da classe `KeyboardEvent`. Esta classe deve implementar apenas dois métodos obrigatórios (`__hash__` e `__eq__`) que são usados para guardar os eventos num dicionário, ficando cada evento associado à sua ação. Opcionalmente pode ser definido um terceiro método, `get_parameters`, que deve returnar um dicionário com as variáveis e os seus respetivos valores que o evento disponibiliza. Cada tipo de evento pode também definir uma propriedade chamada `binary` como verdadeira ou falsa.

Os **binários** referem-se ao conceito de eventos que são conceptualmente compostos por duas fases: premir e soltar. Exemplos deste tipo de eventos são, obviamente premir e soltar teclas do computador ou de um piano. A razão porque este conceito é tratado como um caso particular na nossa linguagem, ao invés de serem tratados como dois eventos separados (`PianoKeyPress` e `PianoKeyRelease` por exemplo), deve-se ao facto de os eventos binários mapearem de forma bastante elegante com o conceito de *note on* e *note off* na geração de música. E da mesma maneira que a nossa linguagem não obriga o utilizador a declarar por cada nota o seu ponto de início e de fim separados, não faria sentido fazer isso para os teclados.

Por essa razão, os teclados podem (através dos modificadores `hold extend` que vamos cobrir a seguir) mapear automaticamente o início e o fim das notas declaradas em cada uma das suas teclas, com os modos *press* e *release* dos seus eventos.

5.3.3.2 Modificadores de Eventos

Cada evento de um teclado tem associada uma ação: essa ação pode ter *size effects*, e opcionalmente returnar um evento músical (ou uma sequência de eventos). Sempre que o evento é despoletado, a ação é avaliada e caso retorne um objeto músical, esse é reproduzido.

Por predefinição, o tempo que a tecla está premida não afeta a duração das notas emitidas. Pelo contrário, a duração das notas (e acordes e todo o resto de eventos músicais) é calculada com o que o utilizador tiver determinado no código. Isto acontece porque o teclado permite que o utilizador defina não só um evento músical para cada tecla, mas sim que possa definir uma música completa, se quiser. Neste caso quando a tecla é premida, a música começa a tocar até terminar.

Os **modificadores** permitem customizar o comportamento do teclado quando encontra alguma sequência músical. Cada modificador pode ser aplicado a todo o teclado (quando aparece à frente da *keyword* `@keyboard`), ou ser aplicada individualmente a cada tecla.

repeat Quando presente, o teclado reproduz a música da tecla, e quando esta acabar, começa a tocar novamente, sem parar.

toggle Reproduz a música da tecla até esta ser premida novamente (ou a música acabar).

hold Reproduz a música da tecla enquanto a tecla estiver premida (ou a música acabar).

extend Tanto o `toggle` como o `hold` permitem terminar uma música mais cedo. Com este modificador, todas as notas tocadas pela tecla ficam ativas enquanto a pessoa não carregar novamente na tecla (em conjunto com o `toggle`), ou a largar (em conjunto com o `hold`).

Assim, se quisermos replicar o comportamento de um piano, por exemplo, em que premir a tecla começa a tocar uma nota, e libertá-la para a nota, podemos usar os modificadores `hold extend` em conjunto.

```
@keyboard hold extend {
   a: c;
   s: d:
   d: e;
};
```

Listing 5.10: Aplicar o modificar `hold extend` a um teclado inteiro

5.3.3.3 Estruturas de Controlo

Até agora temos visto como é possível declarar manualmente ações num teclado. Algo que ainda não foi dito é o facto de o bloco de declaração de um teclado `@keyboard {}` ser uma macro que, antes da execução, é traduzida para instruções que criam o teclado e registam as teclas.

```
{
   $__keyboard = keyboard\create();
   keyboard\push_flags($__keyboard; 'hold'; 'extend');
   keyboard\register($__keyboard; 'a'; c);
   keyboard\register($__keyboard; 's'; d);
   keyboard\register($__keyboard; 'd'; e);
   keyboard\pop_flags($__keyboard; 'hold'; 'extend');
   $__keyboard
}
```

Listing 5.11: Código gerado automaticamente para criação do teclado descrito no capitulo anterior

Vendo desta forma é possível concluir que o código fica extremamente mais verboso, mas ao mesmo tempo abre novas possibilidades: é possível registar teclas condicionalmente, envolvendo a chamada da função num `if`, ou registar teclas em massa dentro de um ciclo `for` ou `while`.

A abordagem de criar os teclados manualmente desta forma é sem dúvida válida para os casos mais complexos (com alguns dos teclados definidos na biblioteca *standard* a serem criados desta forma). Mas achamos que seria interessante poder integrar este tipo de estruturas de controlo simples com a sintaxe

específica de declaração de teclados, para permitir mais variedade ao tipo de teclados que são possíveis de criar com ela.

Assim sendo, dentro da expressão de declaração de teclados, são suportados os seguintes construtores sintáticos:

Condicionais Permitem tornar a declaração de uma ou mais teclas condicionais. O código gerado é transformado de forma a envolver as funções que registam as teclas dentro de um `if`.

Ciclos É possível ter também ciclos `for` e `while` dentro de um teclado. Isto permite percorrer um *array* ou repetir a mesma tecla com alguma variação várias vezes.

Blocos É possível envolver instruções da linguagem em chavetas em qualquer parte dos teclados.

Tomemos um pequeno exemplo que associa a quatro teclas a ação de imprimir para o ecrã um número. Podemos ver que é possível encadear os construtores uns dentro dos outros (neste caso um `if` dentro de um `for`). Mas mais interessante do que isso, é verificarmos que existe um pequeno bloco de código responsável por declarar a variável `$ki = $i`, e que no `print` são impressos as duas variáveis.

```
$i = 0;

@keyboard {
   for ($k in @[ 'a', 's', 'd', 'f' ]) {
      if ($i > 0) {
         { $ki = $i };
         [$k] hold extend: print( $i, $ki );
      };
      { $i += 1 };
   }
};
```

Listing 5.12: Declaração de teclado dinâmica recorrendo ao uso de ciclos, condicionais e blocos de código.

Não seria invulgar pensar que ambos os números seriam iguais sempre que fossem impressos, mas isso não é o caso. Isto providencia uma boa ocasião para reforçar a noção que as teclas ficam associadas a expressões, e essas expressões são executadas só quando a tecla é ativada.

Isto significa que quando qualquer tecla for premida, o valor de `$i` será sempre o mesmo: 4. Isto porque a variável está declarada fora do teclado, e mais importante, fora do ciclo `for`, o que significa que todas as teclas referenciam a mesma variável. Por outro lado, a variável `$ki` é declarada pela primeira vez dentro do ciclo, e como tal está a criar quatro símbolos diferentes que são depois cada um referenciado pela tecla respetiva.

5.3.3.4 Gravar e Reproduzir Performances

5.3.3.5 Buffers

Ao contrário de performances, que gravam os eventos que acionam as teclas dos teclados, é possível guardar diretamente os eventos músicais produzidos pelos teclados com a class `keyboard\Buffer`. Os eventos capturados em *buffers* podem ser tratados como qualquer outra sequência músical, e como tal, ser composta com outras expressões, passada a funções e tudo o resto que a linguagem possibilita.

```
1 $buffer = keyboard\Buffer( $keyboards = ..., $start = true );
```

Listing 5.13: Instanciação de um *buffer*

Ao construir um *buffer*, por predefinição, todos os teclados são gravados imediatamente. É no entanto possível passar uma lista a limitar quais os teclados que irão ser gravados para o buffer, útil para permitir dividir as notas gravadas em faixas separadas (um *buffer* grava uma faixa, outro grava uma faixa diferente, etc). Para além disso, é possível também não iniciar a gravação para o buffer imediatamente.

Entre as funcionalidades disponibilizadas pela classe, podemos destacar as mais importantes como sendo:

start Começa a gravar os eventos emitidos. Mesmo depois de chamado o método `start()`, o *buffer* só inicia a gravação quando for capturado o primeiro evento músical. Caso se queira forçar um *padding* silencioso ao início, pode-se emitir um evento de pausa com duração zero. Caso o buffer já tenha conteúdos em memória, o que for gravado agora é acrescentado ao fim.

stop Para a gravação do *buffer*. Mais uma vez, a gravação é cortada no último evento músical. Caso se queira forçar a duração do *buffer* com mais algum tempo depois da última nota, basta emitir um evento de pausa com duração zero quando realmente quisermos que o *buffer* termine.

clear Limpa tudo o que o buffer tiver gravado e permite reutilizar o *buffer* do ínicio.

to_music Retorna os conteúdos do *buffer* na forma de uma sequência músical. Esta sequência é ímutável: alterações aos conteúdos do *buffer* não são refletidos nos dados da sequência gerada. Para obter essas alterações, este método deve ser chamado novamente, returnando uma nova sequência músical.

from_music Permite substituir manualmente o conteúdo do *buffer* com uma sequência músical. É útil como iremos ver mais à frente para permitir as operações de `save()` e `load()` dos *buffers*.

Tudo isto permite manipular os *buffers* através da linguagem de *scripting*, mas sendo o objetivo dos *buffers* serem usados em conjunto com os teclados, primariamente quando o utilizador os estiver a usar para compor arranjos músicais, é desejável associar todas estas chamadas de código a teclas do teclado, para as permitir usar durante a sua utilização.

Tendo acesso às funções dos *buffers*, cada utilizador pode então programar a sua utilização como bem quiser. Mas para os casos mais comuns, a biblioteca da linguagem fornece já funções destinadas a facilitar o seu uso.

```
keyboard\bufslot( $bf = keyboard\Buffer( start = false ), $key = "p" );

keyboard\bufpad( ref $buffers, $keys = @[ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' ],
    ↪ $savename = "buffers.mkl", $load_key = 'f7', $save_key = 'f8' );
```

Listing 5.14: Funções disponibilizadas para criação de *buffers* controlados por teclados.

A primeira função `keyboard\bufslot` permite criar apenas um *buffer* associado a uma tecla.

A segunda `keyboard\bufpad` função permite uma utilização mais avançada, associando N *buffers* distintos a N teclas. Para além de permitir iniciar e parar de gravar, bem como reproduzir os conteúdos de cada *buffer*, também permitem gravar e carregar os conteúdos dos *buffers*.

Quando o utilizador pressiona a tecla para gravar ou carregar os *buffers*, é mostrado ao utilizador uma *dialog* para escolher o caminho onde guardar. O caminho introduzido pelo utilizador fica guardado em memória e é preenchido automaticamente da próxima vez que a *dialog* for mostrada, evitando obrigar o utilizador a escrever o caminho de todas as vezes.

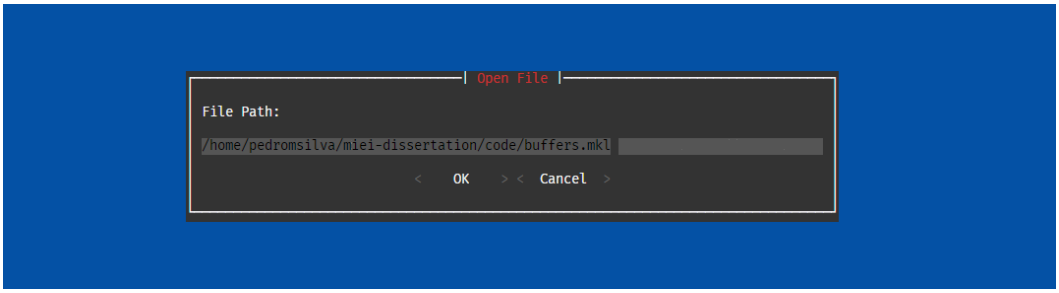


Figure 5.1: Janela de carregamento dos buffers

Os conteúdos do *buffer* são guardados num ficheiro `.mkl`, o que é bastante interessante pois permite evitar ter de criar e implementar algum formato específico para guardar os seus conteúdos. Mas também permite, sendo um formato de texto, e sendo a sua sintaxe a da nossa linguagem *Musikla*, o utilizador pode gravar os *buffers*, abrir o ficheiro com algum editor de texto e efetuar alterações manualmente, e carregar essas alterações de novo para os *buffers* com extrema facilidade.

```
$buffers = @{};
$buffers::set(1; (:default r2/23 [B^d^f]1/29 r1/32 [A^ce]1/32))
```

Listing 5.15: Formato do ficheiro de gravação dos *buffers*.

O formato do ficheiro passa pela declaração de uma variável **$buffers** contendo um dicionário. O dicionário contém depois um par chave-valor para cada *buffer* não vazio aquando da gravação do ficheiro. A chave representa o índice do *buffer*, e o valor representa a sequência musical lá guardada nesse momento.

Para guardar o ficheiro, é criada é memória a sua AST respetiva. Esta é depois convertida em texto através da classe `CodePrinter`.

O processo de carregar o ficheiro é igualmente simples, passando por executar o *script* num contexto isolado. Depois o valor da variável `$buffers` é obtido, e as entradas do dicionário são percorridas, substituindo os valores presentes nos *buffers* recorrendo á funcção `keyboard\Buffer.from_music()`.

5.3.4 Grelhas

5.3.5 Transformadores

5.3.6 Editor Embutido

Chapter 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 aúdio. 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 músical 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 linguage *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.

.1 Apendíce 1: Gramática

```
main <- body EOF;

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

// Statements
statement <- _ ( var_declaration / voice_declaration / function_declaration /
    ↪ for_loop_statement / while_loop_statement / if_statement / expression ) _;

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

voice_declaration <- ":" identifier _ "=" _ voice_declaration_body;

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

function_declaration <- "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_declaration <- "@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 <- "^" / "^^" / "__" / "_" / "";
```

```
84
85  note_pitch
86  <- r”[cdefgab]” ”'”*
87     / r”[CDEFGAB]” ”,”*
88     ;
89
90  modifier
91  <- r”[tT]” _ integer
92     / r”[vV]” _ integer
93     / r”[lL]” _ note_value
94     / r”[sS]” _ integer _ ”/” _ integer
95     / r”[sS]” _ integer
96     / r”[oO]” _ integer
97     ;
98
99  instrument_modifier <- ”:” ( identifier / ”?” ) _ sequence;
100
101 value_expression <- expression expr_binary_op expression
102             / expr_unary_op expression
103             / string_value
104             / number_value
105             / bool_value
106             / none_value
107             / variable
108             / function
109             / keyboard_declaration
110             / group
111             / block
112             / chord
113             / note
114             / rest
115             / modifier
116             / instrument_modifier
117             ;
118
119 exp_binary_op <- ”and” / ”or” / ”>=” / ”>” / ”==” / ”!=” / ”<=” / ”<” / r”[+\-*/]”;
120
121 expr_unary_op <- ”not”;
122
123 string_value <- double_string / single_string;
124
125 double_string <- ”\”” double_string_char* ”\””;
126
127 double_string_char
128 <- ”\\\””
```

```
    / "\\\\"
    / 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]*";
```