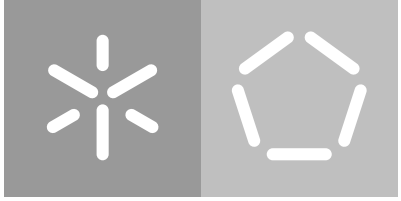
Universidade do Minho
Escola de Engenharia

Pedro Miguel Oliveira da Silva

Musikla
Music and Keyboard Language

Agosto, 2020



Universidade do Minho
Escola de Engenharia

Pedro Miguel Oliveira da Silva

Musikla
Music and Keyboard Language

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efetuado sob a orientação do(a)
José João Dias de Almeida

Agosto, 2020

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Agradecimentos

Quando estamos perante um momento importante da nossa vida, é fácil pensarmos nele como um momento circunspecto e isolado. Uma fase que acaba e outra que começa. Um antes e um depois. Mas a minha experiência até agora mostrou-me que na verdade, a minha vida tem sido mais um contínuo. E nesse contínuo estão um conjunto de pessoas sem o qual seria impossível pensar sonhar em fazer o que fiz.

Por isso mesmo, em primeiro lugar, gostava de agradecer profundamente ao meu orientador de dissertação, o Professor José João Almeida, pela ajuda durante todo este processo. Sempre pronto a transmitir experiência e conhecimento, novas ideias, críticas construtivas e positivas, incentivando-me a agarrar todas as oportunidades, mas mais importante que tudo, encorajando-me sempre a seguir ao meu próprio ritmo e tomar o meu próprio caminho.

Gostava também de agradecer aos meus colegas de curso, em especial à Sofia Silva, sempre pronta a ajudar para qualquer dúvida que eu tivesse e para me dar motivação para seguir em frente.

Também não me posso esquecer de agradecer a todos os professores que tive, em particular aos professores Carlos Carvalho, Luís Cerejeira, Daniel Rego e Pedro Ribeiro, que me ensinaram, inspiraram, e se esforçaram para me providênciaram oportunidades e experiências que nunca irei esquecer, e que me abriram portas na vida que de outra forma me estariam fechadas.

E mais importante que tudo, gostava de agradecer à minha família. À minha mãe, que sempre me apoiou durante todo o curso e fez o que fosse preciso para tornar a minha experiência o melhor possível. À minha tia, uma das pessoas mais trabalhadoras que conheço, que não pensa duas vezes antes de me ajudar em tudo o que eu precisar, e que está sempre presente para me apoiar. E finalmente ao meu irmão, a quem eu sempre quis seguir os passos, e que me fez querer começar a programar desde os doze anos de idade. Se não fosse por ele, não estaria a fazer esta dissertação, nem a acabar este curso, nem teria tido as oportunidades que tive.

A todos, estou eternamente grato. Muito obrigado.

Resumo

Título: Musikla - Linguagem para Música e Teclados

Nesta dissertação iremos estudar uma abordagem para a análise, criação e descrição de música através de uma Domain Specific Language (DSL). Trata-se de uma linguagem dinâmica, com todas as funcionalidades a que estamos habituados, tais como variáveis, funções, ciclos, condicionais. Para além disso, os dois fatores de diferenciação passam pela sintaxes especializadas para declaração de acompanhamentos musicais e de teclados virtuais. Esta linguagem deve depois poder ser avaliada e os seus resultados convertidos para diversos formatos, desde ficheiros de som, MIDI, ou reproduzir diretamente as notas para as colunas do computador.

Com este intuito vamos analisar as linguagens já existentes neste espaço, bem como quais as funcionalidades que já implementam, e aquelas que consideramos estarem em falta.

Como esses aspetos em mente, de seguida propomos uma linguagem que tente aproveitar as boas ideias daquilo que já existe, mais as nossas soluções para os novos desafios que encontramos. Introduzimos também vários casos de estudo para demonstrarem as vantagens que acreditamos existirem na nossa abordagem.

Finalmente descrevemos também o processo de desenvolvimento da linguagem, dividido em três fases principais:

1. O desenho da sintaxe, da sua gramática, e do *parser*.

2. A implementação do interpretador.

3. O desenvolvimento de uma biblioteca *standard* para ser incluída com a linguagem.

A nível da sintaxe e da gramática, descrevemos sucintamente toda a linguagem. Damos particular atenção às expressões de declarações de acompanhamentos musicais e de teclados. Em termos gramaticais, são apenas expressões, ou seja, as suas sintaxes devem integrar-se homogeneamente no resto da linguagem. E como tal, podemos utilizá-las em qualquer sítio que onde podemos introduzir uma expressão, seja ela um número, uma *string* ou o que quer que for.

Esta integração sem separação significa que todos os aspetos da linguagem têm de ser pensados de forma a coexistir sem problemas. Iremos por isso analisar quais os desafios encontrados pela introdução

destas novas classes de expressões na gramática, e quais as soluções que foram tomadas para contornar essas situações.

A nível do interpretador, discutimos várias das opções que poderiam ser escolhidas (interpretadores *tree walk*, máquinas *bytecode*, compilação JIT) bem como justificamos a nossa escolha de utilizar um interpretador *tree-walk*.

A nível da biblioteca *standard*, descrevemos os vários formatos suportados, quer de *input*, quer de *output*, bem como os mecanismos providenciados para a utilização de teclados, como grelhas e *buffers*.

No final, descrevemos como correr scripts escritos na nossa linguagem: através de uma aplicação de linha de comandos desenvolvida em *Python*, chamada **musikla**, publicada no Python Package Index (PyPI)[1], e cujo código é disponibilizado livremente no *GitHub*[2].

Palavras-chave: Interpretador, Linguagem de Domínio Específico, Notação Musical, Processamento de Linguagens...

[1] https://pypi.org/project/musikla/

[2] https://github.com/pedromsilvapt/miei-dissertation

Abstract

Title: Musikla - Music and Keyboard Language

In this dissertation we'll study an approach to the analysis, creation and description of music through a Domain Specific Language (DSL). It is a dynamic language with all the features we are used to, such as variables, functions, loops and conditionals. Furthermore, the two diferrentiating factors about this language are the specialized sintax for declaration of musical arrangements and virtual keyboards. Once evaluated, the results of this language whould be able to be converted into multiple formats, ranging from sound files, MIDI files, our even sounds played directly by the computer's speakers.

To accomplish that, we'll analyze existent languages in this space, as well as what functionalities they already implement, and which ones we consider missing from them.

With those aspects in mind, we'll start by proposing a language that tries to reuse the good ideas that are already in use by other projects, plus our own solutions to the challanges we find. We'll also list several case studies that demonstrate what we believe are the main advantages in our approach.

Finaly we'll describe as well the process of developing said language, divided in three main phases:

1. The design of the syntax, its grammar and parser.

2. The interpreter's implementation.

3. The development of a standard library to be included in the language.

With regards to the syntax and the grammar, we'll briefly describe the entire language, giving particular attention to the musical arrangements and keyboards' declaration expressions. Gramattically, those are regular expressions, and so their syntaxes must integrate seamlessly in the rest of the langfuage. This means being able to use them anywhere we could use an expression, be it a number, a string our anything else.

This integration without any specific separation means that all the aspects of the language must be thought of in a way to coexist without issues. Because of that, we'll discuss the challanges we faces by introducing these new classes of expressions in our grammar, and what solutions we found to go around those situations.

In terms of the interpreter, we discuss several options that could be chosen (tree-walk interpreters, bytecode machines, JIT compilation), as well as the justification for our ultimate choice of building a tree-walk interpreter.

In terms of the standard library, we describe the multiple formats supported, both for input and output, as well as the provided facilities to the use of our virtual keyboards, such as grids and buffers.

In the end, we describe briefly how to run scripts written in our language: through a command line application developed in *Python*, called **musikla**, published in Python Package Index (PyPI)[3], and whose source code is freely available on *GitHub*[4].

Keywords: Domain Specific Language, Interpreter, Language Processing, Music Notation ...

[3] https://pypi.org/project/musikla/

[4] https://github.com/pedromsilvapt/miei-dissertation

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Glossário

acidental	Corresponde à nota de uma frequência que não pertence à escala musical atual. Representam-se pelos símbolos sustenido ♯ (*sharp* em inglês), bemol ♭ (*flat*) ou bequadro ♮ (*natural*), e que geralmente aumentam ou diminuem a frequência da nota por um semitom.
acorde	Conjunto de várias notas que são geralemente tocadas em simultâneo.
arpeggio	Passa pela reprodução sequêncial das notas de um acorde (em vez de as tocar em simultâneo). Pode ser simples (tocar cada nota uma vez em sequência) ou seguir uma melodia mais complexa usando as notas disponíveis.
escala diatônica	Uma escala música composta por sete notas, como um intervalo de 12 semitons entre as suas notas. Este padrão de notas repete-se a cada oitava nota, subindo ou descendo a sua frequência.
ligaduras	Símbolo colocado nas partituras músicais para ligar notas seguidas com o mesmo *pitch* em diferentes compassos. As durações das notas são somadas e elas são tocadas como uma só nota.
oitava	Intervalo entre uma nota música, e a correspondente com dobro ou metade da sua frequência. Na escala diatônica, uma oitava corresponde a 12 semitons.
semitom	Também chamado de meio-tom, é o menor intervalo utilizado na escala diatônica. Representa a distância tónica entre duas teclas adjacentes de um piano.
sintetizador	Dispositivo ou *software* responsável por gerar sinais de aúdio em tempo real.

Siglas

AOT	Ahead Of Time
API	Aplication Public Interface
AST	Abstract Syntax Tree
CFG	Context Free Grammar
DAW	Digital Audio Workstation
DSL	Domain Specific Language
HTML	HyperText Markup Language
I/O	Input/Output
IDE	Integrated Development Environment
JIT	Just In Time
LALR	Look Ahead Left Right
MIDI	Music Instrument Digital Interface
PEG	Parsing Expression Grammar
PyPI	Python Package Index
SVG	Scalable Vector Graphics
WAV	Waveform Audio File Format

Publicações

1. Musikla: Language for Generating Musical Events

Autores: Pedro M. Silva e José João Almeida

Livro/Editora: OASIcs, Volume 83, SLATE 2020

Ano: 2020

Abstract: In this paper, we'll discuss a simple approach to integrating musical events, such as notes or chords, into a programming language. This means treating music sequences as a first class citizen. It will be possible to save those sequences into variables or play them right away, pass them into functions or apply operators on them (like transposing or repeating the sequence). Furthermore, instead of just allowing static sequences to be generated, we'll integrate a music keyboard system that easily allows the user to bind keys (or other kinds of events) to expressions. Finally, it is important to provide the user with multiple and extensible ways of outputing their music, such as synthesizing it into a file or directly into the speakers, or writing a MIDI or music sheet file. We'll structure this paper first with an analysis of the problem and its particular requirements. Then we will discuss the solution we developed to meet those requirements. Finally we'll analyze the result and discuss possible alternative routes we could've taken.

Keywords Domain Specific Language, Music Notation, Interpreter, Programming Language

Estado Publicado

Capítulo 1

Introdução

No âmbito das linguagens de programação e engenharia de *software*, é comum observar como a linguagem, ou o paradigma, de programação escolhido pode afetar de modo profundo as decisões que o programador toma, e por consequência, o produto final. De modo análogo, nós acreditamos que a forma como pensamos e falamos sobre música afeta também a música que criamos, ou a forma como pensamos sobre ela. A linguagem musical mais comum utilizada entre profissionais e entusiastas é certamente a linguagem das pautas musicais.

Mas esta linguagem é estática. É como se pudéssemos falar de matemática usando apenas números, mas sem nenhuma linguagem comum para expressarmos conceitos de mais alto nível como equações ou inequações. Era certamente possível representarmos os resultados finais, mas perdia-se a estrutura que possibilitava compreender a um mais alto nível o problema, e até mesmo generalizar a sua solução e encontrar padrões para outros problemas similares.

Sendo o processamento de linguagens, em particular de linguagens de programação, uma área de enorme interesse pessoal, surgiu a ideia de como integrar vários desses conceitos na linguagem musical, e desta forna torná-la mais rica e poderosa.

O objetivo desta dissertação irá então passar por desenhar e desenvolver uma linguagem que pode ser pensada como uma calculadora musical. Queremos analisar qual a melhor forma de descrever música em formato textual, e de integrar construções comuns na programação, como variáveis, funções, ciclos e estruturas de decisão, entre outros.

Esta linguagem deve então servir tanto como uma ferramenta de suporte teórico, para os utilizadores estudarem a estrutura e a teoria de peças musicais, mas também como uma ferramenta prática para praticar a criação de música.

Esta última parte prática motivou também a incorporação na nossa linguagem de uma componente de interatividade. Para além de permitir usar pianos, ou o próprio teclado de um computador, para reproduzir as notas habituais, queremos dar ao utilizador a possibilidade de customizar as ações ou os

acompanhamentos musicais que tocam em cada tecla.

Um sentimento que iremos repetir bastante ao longo desta dissertação é a necessidade deste projeto ser extensível. Isto é, permitir a cada utilizador adicionar novas funcionalidades à linguagem quando precisar, e tornar este processo o mais simples possível, sem fases de recompilação ou necessidade de alterar o código fonte do projeto. Isto porque a produção de música é uma área enorme, e mais importante, é uma forma de expressão criativa. Não temos ilusões de conseguir cobrir sequer a maior parte dos casos de uso de todos os utilizadores. E por isso, providenciar uma base sólida sobre a qual cada pessoa pode facilmente construir, e eventualmente ir juntando essas construções à implementação oficial, parece-nos crítico para o sucesso e para a usabilidade da linguagem.

Para realizar isso, iremos analisar no próximo capítulo as linguagens existentes na interseção da programação com a criação de música. Vamos analisar para cada uma, quais os aspetos que não coincidem com aquilo que queríamos encontrar numa linguagem, mas também aquilo que já existe e que está bem feito. E como as boas ideias são para ser aproveitadas, iremos incorporar o que for sensato na nossa linguagem (como por exemplo, usar a sintaxe de descrição de notas e acordes do projeto *abc notation*[1]).

[1] http://abcnotation.com/

Capítulo 2

Estado da Arte

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

2.1 Trabalho Relacionado

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

Language	**Tipo de Dados**	**Dinâmica**	**Interatividade**
alda	Notação Musical	Não	N/A
abc notation	Notação Musical	Não	N/A
Faust	Sinais Áudio	Sim	Botões/Sliders/etc...
SuperCollider	Sinais Áudio	Sim	Não
ChucK	Sinais Áudio	Sim	Teclados Imperativos
SonicPi	Notas	Sim	Não
Solução Ideal	Notação Musical	Sim	Teclados Declarativos

Tabela 2.1: Lista de modificadores e exemplos da sua utilização

Queremos analisar as opções segundo os seguintes conceitos:

Tipo de Dados Refere-se a quais os tipos de dados usados principalmente pela linguagem na componente musical. Sinais de áudio são os de mais baixo nível, enquanto que *Notas* refere-se à linguagem permitir tocar uma notas manualmente. `Notação Musical` é a de mais alto nível e permite descrever não só notas, mas acompanhamentos musicais.

Dinâmica Corresponde a saber se a linguagem tem construções dinâmicas (como variáveis, ciclos, funções) ou se é apenas estática.

Interatividade Algumas linguagens podem permitir interatividade através de interfaces gráficas no computador, como botões e *sliders*. Outras podem suportar teclados, que podemos dividir em dois tipos: *Imperativos*, e *Declarativos*. Imperativos referem-se à possibilidade de ser notificado quando qualquer tecla é premida, uma de cada vez, e decidir manualmente o que fazer. Declarativos referem-se a descrever uma lista de teclas e ações a executar para cada uma delas, sem termos de nos preocupar em receber manualmente os eventos e decidir se o evento se aplica à nossa tecla ou não.

Iremos de seguida analisar mais a fundo estas linguagens, bem como comparar as funcionalidades que cada uma oferece relativamente aos conceitos enumerados em cima.

2.1.1 Alda

O projeto **alda** ("alda", s.d.) 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
```

Listagem 2.1: Exemplo da linguagem alda

É também possível verificar o uso de acidentes (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", s.d.; Gonzato, 2019) é uma notação textual que permite descrever notação musical. É bastante completa, tendo formas de descrever notas, acordes, acidentes, ligaduras de notas, *lyrics*, múltiplas vozes, entre outros.

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

A complexidade da notação traz tanto vantagens como desvantagens, no entanto: A sua ubiquidade significa que uma maior percentagem de utilizadores já se pode sentir à vontade com a sintaxe, o que não acontece com outras linguagens menos conhecidas. Mas por outro lado, conhecer ou implementar toda a especificação ("ABC Notation Standard v2.1", 2011) 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 possíveis, mas os mais usados são: **X** (número de referência), **T** (título), **M** (compasso), **L** (unidade de duração de nota) e **K** (armação de clave).

```
1 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'|]
```

Listagem 2.2: Exemplo da notação ABC

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

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

Listagem 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 acidentes 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 Examples", 2011).

2.1.3 Faust

A linguagem **Faust** (Orlarey, Fober & Letz, 2009; Orlarey, Graef & Kersten, 2006) é uma linguagem de programação funcional com foco na sintetização de som e processamento de áudio. Ao contrário das linguagens analisadas até agora, não trabalha com abstrações de notas e elementos musicais. Em vez disso, a linguagem trabalha diretamente com ondas sonoras (representadas como *streams* de números) e através de expressões matemáticas, que de uma forma funcional permitem 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 Bytecode, instrumentos VST/VSTi). Também permite gerar aplicações *standalone* para as bibliotecas de audio mais comuns já embutidas ("FAUST Targets", s.d.).

A linguagem vem embutida com uma biblioteca extremamente completa ("FAUST Libraries", s.d.) que implementa, entre muitas outras, funções de matemática comuns, filtros áudio e 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", s.d.) e com um tutorial para iniciantes ("FAUST Quick Start", s.d.), 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;
```

Listagem 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` (sendo `no` o *namespace* contendo as funções de geração de ruído, e `noise` correspondendo ao *white noise*). 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 áudio, 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);
```

Listagem 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;
```

Listagem 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 SuperCollider

O projeto **SuperCollider** (McCartney, 2002; Orlarey et al., 2006) é uma plataforma para geração e sintetização de som e música. É composta em parte pela linguagem interpretada **sclang**, focada na componente de áudio, mas com funcionalidades de programação generalizada. Também tem um servidor de áudio *realtime* **scsynth**, que pode ser controlado pela linguagem *sclang*, e que implementa diversas técnicas de geração de áudio otimizadas (permitindo ao utilizador programar também as suas próprias técnicas customizadas através de C++). Também integra um IDE **scide** que disponibiliza um ambiente de edição integrado para todo o ecossistema, bem como ferramentas de ajuda e introdução à plataforma.

É uma linguagem de baixo nível em termos musicais, mas com um grande ecossistema para integrar os mais diversos componentes, desde controladores MIDI a interfaces gráficas. Mas a nível musical, como já referido, foca-se na sintetização e manipulação de ondas de som, sem ter noção de conceitos mais abstratos como notas ou acordes. Tais noções têm de ser manualmente implementadas pelo utilizador, e de uma forma bastante mais verbosa do que o desejável.

2.1.4.1 Exemplos

A linguagem do projeto *sclang* é uma linguagem orientada a objetos mas com aspetos funcionais (como *currying* ou listas em compreensão).

```
1 { [SinOsc.ar(440, 0, 0.2), SinOsc.ar(442, 0, 0.2)] }.play;
```

Listagem 2.7: Declaração de dois canais de aúdio com base em dois osciladores

No exemplo presente na listagem 2.7, é declarada uma função (demarcada pelo par de chavetas) que retorna uma lista com dois osciladores de ondas sinusoidais. O som dos osciladores é depois reproduzido através da chamada da função `play`. De notar que como os osciladores estão dentro de um *array*, isso significa que estamos a gerar múltiplos canais de áudio (dois neste caso), com um oscilador para cada canal.

O oscilador `SinOsc` é apenas um dos geradores de som disponibilizados pelo servidor *scsynth*, também chamados **UGens**. Existem outros, e mais importante, é possível compor esses geradores para criar sons mais complexos.

Um exemplo ainda relativamente simples desse tipo de composição, presente na listagem 2.8 seria usar o gerador *Pan2* que redireciona o som oriundo de outro gerador para dois canais diferentes. A prevalência do som em cada canal (o *pan*) pode ser costumizada por um segundo argumento, com um valor entre -1 e 1, onde -1 emitiria apenas som no canal da esquerda, 1 emitira apenas som no canal da direita, e qualquer valor pelo meio iria produzir uma gradação entre os dois canais, linearmente proporcional a esse valor.

```
1 { Pan2.ar(PinkNoise.ar(0.2), -0.3) }.play;
```

Listagem 2.8: Dividir um gerador por dois canais de forma desigual

É possível criar uma versão mais interessante deste exemplo quando sabemos que os geradores podem ser utilizados não só para gerar som, mas também para servirem de parâmetros a outros geradores. Por exemplo, na listagem 2.9, em vez de passarmos um número literal -0.3 como segundo argumento, podemos passar um oscilador. Desta forma, o som gerado pelo `PinkNoise` irá variar em proporção ao longo do tempo pelos dois canais gerados, em vez de tocar de forma fixa no mesmo.

```
1 { Pan2.ar(PinkNoise.ar(0.2), SinOsc.kr(0.5)) }.play;
```

Listagem 2.9: Dividir um gerador por dois canais de forma desigual

A linguagem é bastante mais complexa, podendo declarar variáveis, objetos, executar funções, estruturas de controlo como condicionais e ciclos, e muito mais. Relativamente à parte musical, contém ferramentas bastante poderosas, mas apenas de baixo nível, com manipulação da música focada em ondas sonoras.

2.1.5 ChucK

A linguagem **ChucK** (Wang, Cook & Salazar, 2015) (Wang, Cook et al., 2003) é outra linguagem de sintetização de áudio digital, similar aos projetos *SuperCollider* e *Faust*. O seu maior fator de diferenciação advém da sua abordagem única e interessante de sincronização de processos concorrentes baseado em unidades de tempo (que os autores do projeto denominaram **strongly-timed**).

Este conceito significa que o utilizador pode definir tempos virtuais associados a quando certas instruções devem ocorrer. Tal garantia torna-se útil quando combinada com o conceito de *shreds* (processos virtuais que podem estar a correr concorrentemente) e que dão a aparência para o utilizador que estão na verdade a correr em paralelo. A máquina virtual por trás da linguagem, responsável por traduzir as instruções em áudio, assegura-se que os *shreds* são **sample-synchronous**, ou seja, cada *sample* (ou amostra) geradas pelos *shreds* e com o mesmo *timestamp* virtual irão ser sempre reproduzidas ao mesmo tempo (mesmo que o processador tenha demorado mais tempo a correr um dos *shreds* do que os outros). Isto é, apesar de as *samples* poderem-se atrasar, a máquina garante que tal acontece de forma sincronizada, sem a possibilidade de criar desfasamento entre os vários *shreds*.

Mais uma vez esta linguagem lida com o conceito de geração de áudio a um baixo nível, e não é portanto muito adequada para a descrição de notação musical. Apesar disso, este projeto apresenta também um fator diferenciador que permite utilizar a sua componente de agendamento temporal de eventos, para permitir interatividade (através de eventos *MIDI* ou do teclado do computador). Mais uma vez, mesmo estas funcionalidades são de baixo nível (como iremos abordar mais em detalhe nos exemplos) mas é agradável saber que já vêm pelo menos incluídas com a linguagem.

2.1.5.1 Exemplos

A operação central da linguagem *ChucK* passa pelo operador => (também chamado de operador *chucking*). A sua semântica pode ser pensada um pouco como a atribuição a variáveis, ou *piping* de dados. Por exemplo, na listagem 2.10 podemos ver a declaração de um oscilador sinusoidal `SinOsc s` que é *chucked* para a variável **dac**. Esta é uma variável especial da linguagem e que representa o dispositivo de reprodução de áudio (as colunas ou auscultadores).

Na linha seguinte podemos ver também que é atribuída à variável **now** o valor de dois segundos: esta é outra variável especial da linguagem, neste caso responsável por controlar o agendamento da execução de código. Ao executar essa instrução, estamos efetivamente a parar a execução da *shred* atual durante dois segundos (ficando o som do oscilador definido na linha atrás a enviar som durante os dois segundos para o dispositivo de áudio reproduzir).

```
SinOsc s => dac;

2::second => now;
```

Listagem 2.10: Reproduzir um oscilador durante dois segundos

Ao fim dos dois segundos, como o programa não tem mais nenhuma instrução, a reprodução de som termina. Se quiséssemos reproduzir som infinitamente, poderíamos mover a instrução de avançar no tempo para dentro de um ciclo, como vemos na listagem 2.11 (criando o que é chamado de *time-loop*, ou neste caso em particular, um *time-loop infinito*).

```
SinOsc s => dac;

while( true ) {
   2::second => now;
}
```

Listagem 2.11: Reproduzir um oscilador infinitamente

Neste caso o nosso *time-loop* não faz nada senão esperar continuamente. Mas a sua utilidade é revelada quando vemos que podemos mudar as propriedades do som gerado ao longo do tempo. Por exemplo, podemos verificar que para além de conectar o oscilador ao dispositivo áudio, também lhe demos um nome **s**. Assim conseguimos usar esse nome para alterar, dentro do ciclo, a sua frequência, por exemplo, de dois em dois segundos, como vemos na listagem 2.12.

```
SinOsc s => dac;

while( true ) {
   2::second => now;
   Std.rand2f(30.0, 1000.0) => s.freq;
}
```

Listagem 2.12: Reproduzir um oscilador, variando a frequência a cada 2 segundos

Neste caso geramos um valor aleatório entre 30.0 e 1000.0, e associamos esse valor à frequência do gerador utilizado. Para já temos apenas agendado a execução do código a cada dois segundos. Mas podemos avançar o tempo por qualquer duração, para um tempo específico, ou até avançar o tempo com a precisão de *sub-samples*.

```
1::second => now;
100::ms => now;
1::samp => now;
.024::samp => now;
```

Listagem 2.13: Exemplos de instruções de avanço no tempo

Mas algo bastante interessante sobre a variável **now** é que pode esperar não só por durações de tempo pré-definidas, mas também por eventos interativos que não sabemos quando irão acontecer. É desta forma que é implementada a interatividade com o teclado do computador, por exemplo, ou até mesmo com portas MIDI.

```
Hid hi; HidMsg msg;

0 => int device;
if( me.args() ) me.arg(0) => Std.atoi => device;

if( !hi.openKeyboard( device ) ) me.exit();
<<< "keyboard '" + hi.name() + "' ready", "" >>>;

while( true ) {
   hi => now;

   while( hi.recv( msg ) ) {
      if( msg.isButtonDown() ) { <<< "down:", msg.key >>>; }
      else { <<< "up:", msg.key >>>; }
   }
}
```

Listagem 2.14: Exemplos de instruções de avanço no tempo

O exemplo é bastante grande e verboso, e vai de encontro à nossa opinião sobre, apesar de a linguagem permitir interagir com teclados, a definição desta interação por parte do programador é de bastante baixo nível. Se quisermos ter ações associadas a diferentes teclas, temos de construir estruturas de controlo para decidir qual das teclas foi premida e o que fazer em cada caso. Então se quisermos reagir não só a uma tecla, mas a uma tecla com modificadores (por exemplo `Ctrl Shift T`) temos de guardar manualmente o estado das teclas premidas e saber quando podemos despoletar a ação. Isto porque parece que neste aspeto, a linguagem segue um padrão de código imperativo, em vez de declarativo.

Para além disso, se quiséssemos associar algumas ações a teclas do computador, e outras a teclas de um piano MIDI, teríamos de manualmente abrir os dois dispositivos, e depois arranjar alguma forma de esperar concorrentemente pelos eventos de ambos (possivelmente usando uma *shred* diferente para cada dispositivo).

Em conclusão, não é uma interface agradável para quem queira fazer algumas rápidas experimentações musicais, principalmente alguém que seja músico e não um programador mais experiente.

2.1.6 Sonic Pi

Possivelmente a linguagem que mais se aproxima do objetivo pretendido com este projeto, **Sonic Pi** (Aaron, 2016) ("Sonic Pi", s.d.) 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, *arpeggios* e escalas, por exemplo). Para além disso permite tocar *samples*, que são ficheiros Waveform Audio File Format (WAV). A linguagem já traz consigo aproximadamente 164 *samples* que podem ser livremente usadas, mas é também possível ao utilizador usar as suas próprias.

As músicas são compostas por **live loops**, que são grupos de sons que podem estar a tocar simultaneamente. Dentro de cada *live loop* o utilizador pode usar a função `play` para tocar notas, `sample` para reproduzir ficheiros WAV, ou *sleep* para avançar o tempo. Para além disso a linguagem suporta, através da função `with_fx` a reprodução de sons com efeitos (como *reverb*, *pan*, *echo* entre muitos outros ("Sonic Pi Fx Cheatsheet", s.d.)).

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

Apesar de todas estas funcionalidades disponibilizadas, existem áreas onde o *Sonic Pi* fica aquém dos objetivos pretendidos para este projeto. Por exemplo, apesar de permitir tanto receber como enviar eventos MIDI, as suas capacidades de Input/Output (I/O) são bastante primitivas. Também não é fácil utilizar o teclado 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). A sintaxe de declaração de notas favorece acompanhamentos gerados proceduralmente, mas é muito verbosa para tocar manualmente acompanhamentos com mais que algumas notas seguidas.

2.1.6.1 Exemplos

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

Listagem 2.15: Reproduzir um *sample* com valores aleatórios

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

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

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

```
6 end
```

Listagem 2.16: 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 milissegundos. A função `scale` gera uma lista com as notas da escala pedida, e a função `choose` escolhe aleatoriamente uma dessas notas para tocar.

2.2 Gramáticas

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

Tomemos como exemplo o famoso problema do *dangling else* ("Dangling else", s.d.) descrito nas duas classe de gramáticas:

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

Listagem 2.17: Gramática

```
1 statement = ...
2   | conditional_statement
3
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 previsto.

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

2.2.2 Resumo

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

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

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

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

2.3 SoundFonts

O formato *SoundFont* foi originalmente desenvolvido nos anos 90 pela empresa *E-mu Systems* para ser usado inicialmente pelas placas de som *Sound Blaster*. Ao longo dos anos o formato sofreu diversas alterações, encontrando-se atualmente na versão 2.04, lançada em 2005 (*SoundFont Technical*

Specification, 2006). 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++", s.d.), WildMIDI ("WildMidi", s.d.) e FluidSynth ("FluidSynth", s.d.).

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(Rossum & Joint, 1995) é constituído por um ou mais bancos (*banks*) (até um máximo de 128). Cada banco pode por sua vez ter até 128 *presets* (por vezes também chamados instrumentos ou programas).

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

```
1   # Cada preset e identificado por um par: (bank, preset number)
2   SoundFont = Dict[ Tuple[ int, int ], Preset ]
3
4   # Cada instrumento e identificado por um inteiro entre 0 e 127
5   Preset = Dict[ int, Instrument ]
6
7   # Finalmente, cada sample e identificada pelo indice de nota (que iremos abordar mais a
        ↪ frente em detalhe, no capitulo sobre sintetizadores)
8   Instrument = Dict[ int, Sample ]
9
10  # Aqui nao se encontram detalhados os tipos Wave nem SampleOptions.
11  Sample = Tuple[ Wave, SampleOptions ]
```

Listagem 2.18: Sistema de Tipos de um ficheiro SoundFont

2.4 Sintetizadores

A biblioteca FluidSynth é um *software* sintetizador de áudio em tempo real que transforma dados MIDI em sons, que podem ser gravados em disco ou encaminhados diretamente para um *output* de áudio. Os sons são gerados com recurso a SoundFonts *SoundFont Technical Specification*, 2006 (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** (Whitehead, 2019), 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", s.d.) 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 atribuído a um programa da SoundFont, e é possível a qualquer momento mudar o programa atribuído a qualquer canal através do método `Synth.ProgramChange`. Caso se tenha carregado mais do que uma *SoundFont*, é possível usar o método `Synth.ProgramSelect`, que permite especificar o `id` da *SoundFont* e do banco do instrumento a atribuir.

A chave (**key**) representa a nota a tocar. Sendo este valor um inteiro entre 0 e 127, é necessário saber como mapear as tradicionais notas musicais neste valor. Para isso, basta colocarmos as *pich classes* e os seus respetivos acidentes *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 ênfase a uma nota relativamente às restantes. De notar que um valor igual a zero é o equivalente a invocar o método `Synth.NoteOff`.

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

Capítulo 3

O Problema e os seus Desafios

Daqui para a frente iremos chamar **Musikla** tanto à linguagem que desenhamos, mas também ao interpretador de referência desenvolvido. Desenhar a nossa linguagem 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 miríade de linguagens de programação, tanto industriais como académicas, que já foram desenvolvidas, pelo que não serão o foco principal deste projeto. Pelo contrário, neste projeto serão focados com mais detalhe os desafios resultantes da integração da componente musical na linguagem.

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

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

3.1 Objetivos

Nós podemos assim sumarizar os objetivos principais para a nossa linguagem da seguinte forma:

- **Declarativa** As sequências musicais devem ser descritas de uma forma declarativa (em vez de imperativa).
- **Dinâmica** Introduzir conceitos matemáticos ou de programação, tais como funções e variáveis, para a área musical.
- **Interatividade** Tornar possível criar teclados interativos diretamente a partir da linguagem, e que integrem facilmente com o resto das funcionalidades disponibilizadas.
- ***Lazyness*** Tornar *lazyness* a predefinição para as sequências músicais, gerando apenas os eventos estritamente necessários quando são necessários.
- **Eventos Variados** As sequências musicais devem poder descrever extratos musicais complexos, contento notas singulares, pausas, acordes, *arpeggios*, vozes, e mais.
- **Múltiplos *Inputs*** Para além de permitir que os extratos musicais sejam descritos na nossa linguagem, também deve ser possível que eles sejam importados ou convertidos de diferentes fontes, como dispositivos ou ficheiros *MIDI*.
- **Múltiplos *Outputs*** Guardar ou escrever para diferentes *outputs*, os eventos que forem gerados pela nossa aplicação.
- **Extensibilidade** Tornar simples o processo de estender e customizar o projeto, sem ser necessário passos como clonar o projeto, recompilar ou modificar o seu código interno.

3.2 Solução Proposta

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

Como exemplo da extensibilidade da linguagem, irá também ser desenvolvido por cima dela uma biblioteca de construção de teclados virtuais que permitem associar a eventos de teclas notas ou sequências musicais, ou mesmo instruções a serem executadas na própria linguagem.

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

A existência deste **contexto** implícito significa que as funções *Python* não podem ser expostas diretamente para a linguagem, mas graças à expressividade do *Python* é possível construir uma *Foreign*

Function Interface que seja simples de usar e que evite que o utilizador tenha de mapear as funções manualmente. Em vez disso, pode simplesmente marcá-las como sendo **context-free** (funções que não têm noção da existência do contexto implícito), e elas serão então tratadas de forma apropriada.

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

3.2.1 Gramática da Linguagem

A gramática completa da linguagem pode ser vista no Anexo A. Mas antes de abordarmos em mais detalhe como irá der desenhada a gramática da linguagem, podemos abordar dois pequenos exemplos que demonstram a geração de notas musicais.

```
V70 L1 T120;

fun melody () {
   V120;

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

fun accomp () {
   V50; sustainoff();

   ^Cm;
   BM;
   AM;
}

# Create the notes from a melody with a piano and accompany it with a violin in parallel
```

```
20 $notes = :piano melody() | :violin accomp();
21
22 # Play the notes twice
23 play( $notes * 2 );
```

Listagem 3.1: Exemplo da sintaxe proposta da linguagem

O desenho da gramática da linguagem é composto 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.

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

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

3.2.1.1 Instruções e Expressões

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

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

Declaração de Funções `fun function_name ( $arg1, $arg2, <...> ) {  }`

Ciclos While `while (<condition>) {  }`

Ciclos For `for ($var in <expr>) {  }`

Condicionais If `if (<condition>) {  } else {  }`

Atribuições a Variáveis `$var = <expr>;`

Chamada de Funções `function_name( <arg1>, <arg2>, <...>, named_arg = <arg_n> );`

3.2.1.2 Extratos Musicais

A gramática de expressões ou extratos musicais tem como base fundamental os seguintes blocos: notas, pausas e modificadores. As notas são identificadas pelas letras A até G, seguindo a notação de *Helmholtz*(“Helmholtz Pitch Notation”, s.d.) 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
```

Listagem 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]+`.

Compasso Define o compasso, que descreve o tipo de batida da música e o comprimento de uma barra na pauta musical. Tem o formato `[sS][0-9]+/[0-9]+`.

Oitava Permite mudar qual a oitava base (por predefinição 4). Tem o formato `[oO][0-9]`.

Instrumento Qual o identificador numérico do instrumento (a começar em um) a utilizar (regra geral, seguindo o *standard* General MIDI). Segue o formato `[iI][0-9]+`.

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

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

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

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

3.2.1.3 Teclados Virtuais

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

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

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

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

# The set_transform function allows changing all events before they are emitted by this
    ↪ keyboard
$keyboard::map( fun ( $k, $events ) => $events + interval( octave = $octave ) );
```

Listagem 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 novamente

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

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

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

Para além disso, será possível controlar muitos mais 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.

```
# The set_transform function allows changing all events before they are emitted by this
    ↪ keyboard
```

```
$keyboard::map( $events => transpose( $events, octave = $octave ) );
# It is possible to synchronize the keyboard with a grid to align the timings of key presses
# and releases to said grid
$keyboard::with_grid( Grid::new( 1, 4 ) );
# We can also simulate key presses and releases programmatically
$keyboard::start( "ctrl+z" );
$keyboard::stop_all();
```

Listagem 3.4: Exemplo da sintaxe proposta da linguagem

3.2.2 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.

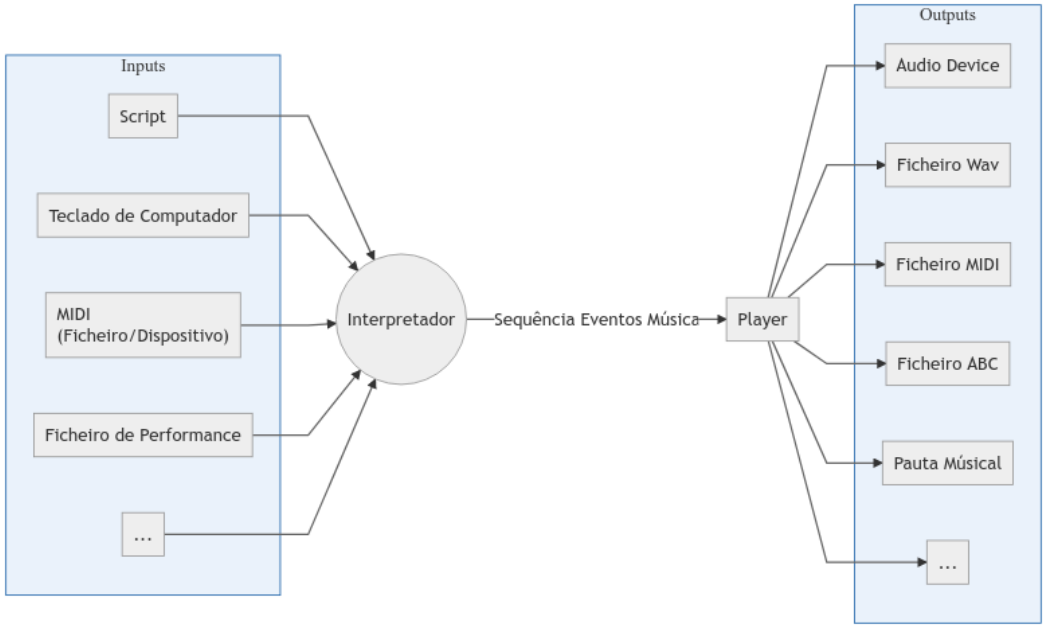


Figura 3.1: Arquitetura Geral do Projeto

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

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

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

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

Eventos A linguagem já disponibiliza uma variedade de eventos multimédia (como reproduzir notas, sons genéricos, ou mensagens para controlar dispositivos MIDI). Mas o objetivo é que criar e emitir eventos 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 musicais, mas também para outros fins como controlar luzes, vídeo e imagens através da linguagem (em conjunto com os eventos personalizados) e permitir assim executar espetáculos multimédia completos a partir da linguagem.

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

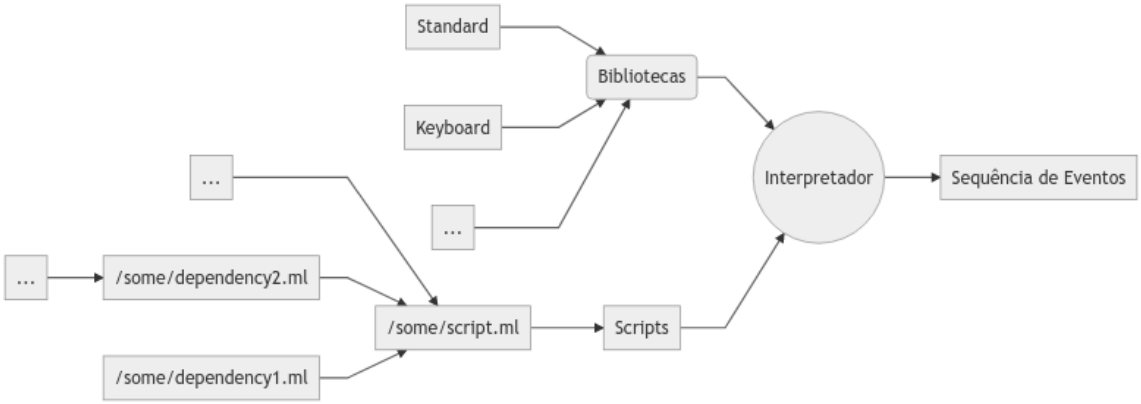


Figura 3.2: Arquitetura do Interpretador

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

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

3.2.3 Álgebra Musical

Dentro da própria linguagem, e acompanhado das construções de programação como ciclos, variáveis, condicionais e tudo o resto, queremos disponibilizar uma *álgebra* que envolva tanto a parte musical como dos teclados e que permita de certa forma manipulá-los. Para isso, a linguagem tem os tipos:

Descrição	Tipo
Sequências de Notas, Acordes, WAVEs, etc	`Music`
Oitavas, tons, semitons	`Interval`
Metadados (compasso, batidas por minutos, etc)	`Voice`
Dicionário de Tecla -> Música ou Função	`Keyboard`

Para operar entre estes tipos, existem um conjunto de operações que são tão comuns que merecem ser implementadas através de operadores sintáticos. Devemos notar que o mesmo operador pode ser usado para ações diferentes com base nos tipos dos seus operandos. Sendo a nossa linguagem dinâmica, este tipo de *overloading* deve acontecer em *runtime*.

Função	Operador
`concat ( Music, Music ) -> Music`	`a b`
`parallel ( Music, Music ) -> Music`	`a | b`
`arpeggio ( Chord, Pattern ) -> Music`	`a * b`
`repeat ( Music, Int | Bool ) -> Music`	`a * b`
`transpose ( Music, Interval | Int ) -> Music`	`a + b`
`retime ( Music, Music | Float ) -> Music`	`a ** b`
`union ( Keyboard, Keyboard ) -> Keyboard`	`a + b`
`exclusion ( Keyboard, Keyboard ) -> Keyboard`	`a - b`

Para além dos operadores disponibilizados, existe um conjunto de funções que permite transformar e compor de maneiras mais avançadas objetos dos variados tipos. Algumas dessas funções (com as assinaturas simplificada relativamente à sua função real, para efeitos de claridade) são listadas de seguida:

Descrição	Função
Leitura de MIDI	`readmidi( File | Port ) -> Music`
Gravação de uma *performance*	`keyboard\record( File )`
Reprodução de uma *performance*	`keyboard\replay( File )`
Leitura de uma *performance*	`keyboard\readperf( File ) -> Music`
Gravação de Música	`save( Music, File... )`
Transformar Teclados	`map( Keyboard, Fn ) -> Keyboard`
Filtrar Teclados	`filter( Keyboard, Fn ) -> Keyboard`
Transformar Música	`map( Music, Fn ) -> Music`
Filtrar Música	`filter( Music, Fn ) -> Music`
Cortar Música	`slice( Music, Start, End ) -> Music`

Capítulo 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 pelo interpretador de referência desenvolvido nesta dissertação. 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. Estes casos de estude não são apenas uma forma de mostrar como os vários componentes da aplicação funcionam em conjunto, mas são também uma forma de especificação da linguagem. É muitas vezes mais fácil decidir qual a melhor forma de implementar uma funcionalidade quando temos vários casos de estudo de qual é a forma que podemos tirar melhor proveito da mesma.

4.1 Tocar Música

```
1  # Title: Westworld Main Theme
2
3  :piano = I1 S6/8 T70 L/8 V120;
4  :violin = :piano(I41);
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 );
```

Listagem 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 Fuga de Duas Vozes

O caso de estudo anterior mostra como é possível declarar algumas notas, e aproveitar operadores simples como a repetição para reduzir a redundância no código. Neste caso de estudo usamos conceitos mais avançados e vemos como podemos declarar uma função, que usa as suas variáveis em mais do que um sítio e de diferentes formas para produzir o resultado final.

```
1  # Fugue 2 in C minor in Book I of the J.S. 'Bachs Well-Tempered Clavier
2  fun fugue ( $subj, $resp ) =>
3     ( $subj $resp | r ** $subj ( $subj + 7 ) );
4
5  S8/4 T140 L/4 V120;
6
7  $subj = r c/2 B/2 c G _A c/2 B/2 c d
8        G c/2 B/2 c d F/2 G/2 _A2 G/2 F/2;
9
10 $resp = _E/2 c/2 B/2 A/2 G/2 F/2 _E/2 D/2 C _e d c
11       _B A _B c ^F G A ^F;
12
13 play( fugue( $subj, $resp ) );
```

Listagem 4.2: Exemplo da declaração da estrutura e conteúdo de uma simples fuga de duas vozes

Também nos permite analisar que as músicas podem conter estruturas não imediatamente óbvias à primeira vista. A função recebe duas excertos musicais `$subj` e `$resp`, e toca-os em sequência. Mas para além disso, durante o terceiro e quarto compassos, também toca em paralelo uma versão do `$subj` transposta 7 semitons.

E este é apenas um pequeno exemplo do tipo de transformações e operações que podem ser efetuadas de modo não destrutivo sobre os acompanhamentos musicais. Se quisermos mudar alguma nota em `$subj`, quando reproduzirmos o ficheiro novamente, essa alteração irá refletir-se automaticamente na versão transposta também.

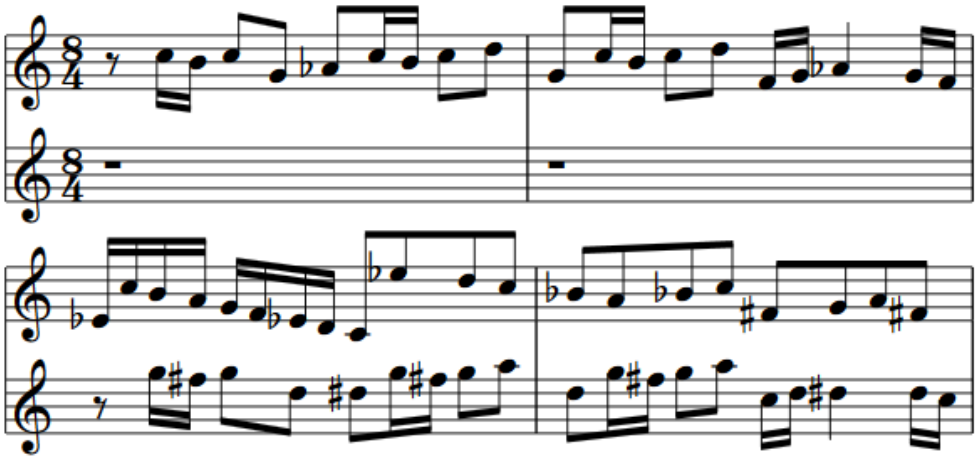


Figura 4.1: Pauta musical gerada pela linguagem, versão áudio disponível aqui[1].

4.3 Definir um teclado

No início do capítulo 4.1 e 4.2 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 then 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 );
};
```

Listagem 4.3: Exemplo da sintaxe para criação de teclados

[1] Fuga https://drive.google.com/file/d/1dIfvnhhKn73Vpp0W6ss6RLsv6PQ_HFTF/view

A sequência de teclas a ser premida neste exemplo é: **A 5 5 5 4 S 3 2 1 D S A 5 6 7 D 7 7 6 S 6 9 8 9 6 7**. Os tempos de cada tecla são relativamente uniformes e podem ser facilmente deduzidos ouvindo a versão sonora deste extrato musical, disponível aqui[2].

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 na sub-secção 3.2.1.3).

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

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

4.4 Teclado de *Arpeggios*

Neste caso vamos construir um teclado mais dinâmico. Vamos associar a algumas teclas diferentes acordes. Mas em vez de tocar simplesmente esses acordes, vamos tocá-los como arpeggios aplicados a um padrão.

É importante notar que o padrão não está escrito estaticamente, mas sim guardado na variável `$pat`. Isto significa que podemos depois usar um segundo *keyboard* para controlar qual o padrão ativo. E para tornar o exemplo ainda mais divertido, usamos as setas *up* e *down* para aumentar ou diminuir a transposição das notas resultantes, dando assim mais variedade aos quatro acordes definidos.

É fácil de ver também como este teclado é extremamente versátil, e pode ser facilmente adaptado para um qualquer número de acordes e *arpeggios* que se queira utilizar.

[2] Soft to Be Strong https://drive.google.com/file/d/1ncy4vlCbbiGQ14lurfsU4LCCUBK0ve13/view?usp=sharing

```
S4/4 L/4 T132 V120;

$patterns = @[
   C D E D2 E C2;
   C r d' e2 r c2;
];

$t = 0; $pat = $patterns::[ 0 ];

@keyboard {
   a: [CM];
   s: [Am];
   d: [FM];
   f: [Dm];
}::map(fun($k, $m) => $m * $pat + $t);

@keyboard {
   1: { $pat = $patterns::[ 0 ] };
   2: { $pat = $patterns::[ 1 ] };

   up: { $t += 1 };
   down: { $t -= 1 };
};
```

Listagem 4.4: Definição de um teclado de acordes

4.5 Teclado Geral com MIDI

Os dois teclados vistos até agora são interessantes quando temos um sub-conjunto específico de notas e acordes com os quais queremos tocar. Mas às vezes é interessante ter uma ferramenta mais generalizada para experimentar.

Este exemplo faz uso de funções disponibilizadas na biblioteca *standard* da linguagem para fazer isso mesmo.

- `keyboard\piano()` Cria um teclado a simular um piano com as teclas do computador. A primeira linha de teclas corresponde às teclas brancas, a segunda às teclas pretas e a terceira novamente às brancas mas uma oitava acima.
- `keyboard\midi()` Cria um teclado que está à escuta dos eventos MIDI (possivelmente de um piano externo ligado ao computador).
- De seguida também vemos que é possível fazermos *override* a algumas teclas se quisermos algo mais específico. Isto é opcional e pode ser ignorado se quisermos.

- `keyboard\bufpad()` Aqui criamos um teclado que está à escuta das teclas numéricas (de 1 até 9) e cria um *buffer* em cada uma delas. Podemos usar esses *buffers* para guardar em memória melodias que toquemos com os teclados, para depois as reproduzir quando quisermos.

- `keyboard\repl()` Finalmente criamos um último teclado que faz a tecla "\"ativar o editor embutido. Assim conseguimos alterar o nosso programa enquanto ele está a correr, alterar teclas, mexer nos *buffers*, declarar variáveis ou gravar os resultados das nossas experiências para disco.

```
# Use a "standard" keyboard piano
$piano = keyboard\piano();
# Maybe even add a MIDI piano?
$piano += keyboard\midi();
# Or declare a custom piano
$piano += @keyboard hold extend {
   a: [^Cm]; s: [BM]; d: [AM];
   f: [EM]; g: [^Fm];
};

# Creates a keyboard that declares a set of buffers. By default,
# the numpad keys are used (creating 10 available buffer slots)
# Save key is F8, Load key is F7
$piano += keyboard\bufpad();
# Creates a keyboard that associates the interpreter accessible by the "\" key
$piano += keyboard\repl();
```

Listagem 4.5: Exemplo da sintaxe proposta da linguagem

4.6 QWERTY Keyboard

No exemplo anterior utilizamos as funções fornecidas pela biblioteca *standard*. Aqui vamos ver um exemplo de como poderíamos programar as nossas próprias funções, utilizando funcionalidades mais dinâmicas (como ciclos `for`) do que simplesmente declarar cada tecla manualmente.

Neste caso estamos a criar um teclado em que cada linha de teclas começa uma oitava acima de onde a linha de baixo começou, e as teclas na mesma linha sobem um semitom cada uma.

```
fun qwertyboard () {
   # Small shortcut function
   fun ivl ($o, $s) => interval( octaves = $o, semitones = $i );

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

```
9        "zxcvbnm,."::split()
10    ];
11
12    $octave = 0;
13    $semitone = 0;
14
15    $keyboard = @keyboard hold extend {
16       for ( $oct, $chars in enumerate( $lines ) ) {
17          for ( $sem, $c in enumerate( $chars ) ) {
18             [ $c ]: c + ivl( -$oct, $sem );
19          };
20       };
21    }::map( fun( $k, $events ) => $events + ivl( $octave, $semitone ) );
22
23    $keyboard += @keyboard {
24       up: { $octave += 1 };
25       down: { $octave -= 1 };
26       right: { $semitone += 1 };
27       left: { $semitone -= 1 };
28    };
29
30    return $keyboard;
31 }
```

Listagem 4.6: 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*, e 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 `$c` corresponde a cada item da *linha* (neste caso, casa tecla), a variável `$sem` permite-nos obter o índice da letra atual na *linha*. A cada letra é associada a nota `c` transposta pelo índice da tecla e da linha a que pertence.

Para além disso também podemos ver a declaração de mais quatro teclas (correspondentes às quatro setas do teclado) que permitem deslocar as notas tocadas por oitavas completas ou por semitons. Para isso, estas teclas têm associadas uma expressão de bloco (identificada pelas 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 seria possível que uma tecla fizesse mais que uma coisa (alterar o estado e no fim retornar ainda notas para serem tocadas, por exemplo).

Finalmente vemos também um exemplo do método `map`, um dos vários métodos que o objeto `Keyboard` disponibiliza e que permitem modificar ainda mais o comportamento dos teclados, alterando o valor emitido por cada tecla de acordo com a função passada.

4.7 Conclusão

Com estes exemplos pudemos ver como ter uma sintaxe declarativa torna bastante compacta a descrição de acompanhamentos musicais. Mas mais do que poder descrevê-los estaticamente, o facto de podermos repetir e transformar sequências musicais aumenta ainda mais a produtividade do utilizador.

Para além disso também conseguimos observar que a integração dos teclados abre portas para situações bastante interessantes, principalmente quando usados em conjunto com as ferramentas de programação dinâmica disponíveis. Um exemplo disso é o caso do nosso teclado de *arpeggios*, onde podemos definir apenas um conjunto de acordes, e depois aplicar uma transformação comum a todos eles para produzirem expressões mais complexas (que podem ser depender de variáveis que vão mudando ao longo da execução).

Para além disso, também concluímos que é importante que a biblioteca *standard* inclua ferramentas genéricas e prontas a utilizar, tais como teclados com as teclas já preenchidas para os casos de uso mais comuns. Isto é útil para facilitar a utilização da linguagem por pessoas que possam sentir-se à vontade com pequenos *scripts* que mudem algumas variáveis e chamem algumas funções, mas não tenham experiência suficiente para criar tudo de raiz.

Por fim, é também algo importante permitir a execução de código (nem que seja apenas de expressões ou instruções singulares, uma de cada vez) em *runtime*. Isto permite ir refinando o programa, principalmente os teclados que possam estar ativos, sem ser necessário terminar a execução do programa. Tal ação, para além de ser mais incomodativo e diminuir a capacidade do utilizador, também apresenta a desvantagem de fazer a aplicação perder a memória, e como tal o estado das variáveis e dos *buffers* que pudessem estar a ser usados.

Capítulo 5

Desenvolvimento do Interpretador Musikla

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 áudio.

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) impõem 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 influenciado 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, afeta 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 linguagem e da aplicação.

5.1 Análise Sintática

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 fase 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 Descrição Sintática

A sintaxe da linguagem é fortemente 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 à componente músical da linguagem. Conseguir juntar estes dois mundos traz consigo alguns desafios que serão discutidos mais em detalhe em cada umas das secções seguintes.

5.1.1.1 Constantes ou Literais

Literais referem-se ao conceito de sintaxe desenhada com o propósito de descrever dados (literais) 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 e as restantes, foi a adição de literais responsáveis por modelar conceitos musicais, como notas, pausas e acordes. Esta sintaxe, tal como já foi mencionado anteriormente, foi inspirada pelo projeto *abc notation*, com algumas modificações.

5.1.1.2 Notas e Pausas

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

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

Listagem 5.1: Expressão Regular que identifica uma nota (quebras de linha adicionadas 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 maiúsculas por minúsculas. Para subir ainda mais oitavas, podemos acrescentar uma ou mais pelicas `'`. Para subir ou descer semitons, podemos prefixar as notas com os acidentes `^` e `_`, respetivamente.

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 parênteses retos. A notação usada para cada nota inclui os seus três primeiros componentes (acidentes, *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 parênteses 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. Esta sintaxe irá suportar mais tipos no futuro, sendo atualmente possível usar os seguintes sufixos:

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

Tabela 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 parênteses retos, a ambiguidade deixa de existir, sendo óbvio que `[C7]` é um acorde de sétima, e `C7` é uma nota com duração 7.

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

Listagem 5.2: Exemplos de três definições de acordes possíveis

5.1.1.4 Modificadores de Voz

Para além de permitir descrever notas, também é possível ter modificadores de voz 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 substituírem simplesmente os valores predefinidos, eles "complementam-se".

Ou seja, se declararmos 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 $4-1=3$) (nota que aqui estamos a falar de oitavas indexadas a zero. No mundo da música elas são geralmente indexadas a um).

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 desse tipo é aplicado).

Nome	**Modificador**	**Exemplo**	**Predefinição**
Instrumento	IN	I46	I1
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

Tabela 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 exceção para as variáveis que possam ter nomes que conflitam com outros construtores sintáticos, seguimos o exemplo de outras linguagens como *PHP*, *Perl* ou *Powershell*, e decidimos prefixar as nossas variáveis com o carácter `$`.

No caso das funções foi possível evitar a ambiguidade (e do mesmo modo a obrigatoriedade de as prefixar com um carácter) devido ao facto de as funções terem obrigatoriamente um par de parênteses (sem espaço entre os mesmos e 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)`, quantos argumentos podemos concluir que a mesma tem? Quatro? 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 na nossa gramática, 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 isso resolveu os problemas de ambiguidade, tornou-se óbvio à medida que que a linguagem foi avançando, que a prevalência da vírgula como separador de argumentos 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, poderíamos 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 Gramática

Aqui vamos analisar brevemente uma versão simplificada da gramática (a versão completa usada pela aplicação pode ser vista no Apêndice A). Nesta versão simplificada, incluída para facilitar a leitura, não aborda detalhes como recursividade à esquerda, ou precedência de operadores. Obviamente, na gramática real foram usadas as técnicas usuais para cobrir esses casos. A notação da linguagem Musikla cobre várias sub-gramáticas, tanto da parte musical, descrição de teclados mas também instruções e expressões de uma programação de linguagem genérica, que iremos abordar de seguida.

5.1.2.1 Corpo

Comecemos pela descrição geral do corpo de um ficheiro Musikla. O corpo é composto por uma lista de *statements* (ou instruções) que são obrigatoriamente separados por pontos e vírgula (sendo apenas o último opcional).

No fim de todas as instruções, opcionalmente, o utilizador também pode inserir um bloco de código *Python* (que é denotado pelo prefixo `@python` e consome todo o texto até ao fim).

```
main <- body python? EOF

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

python <- "@python" python_body

python_body <- ( r".*" r"\r?\n"? )*
```

Listagem 5.3: Produções base da gramática

Ao longo da gramática são também usadas produções triviais, como inteiros, *strings*, booleanos entre outras. Sendo tão comuns e simples, não vale a pena estar a descrevê-las todas. Deixamos no entanto aqui duas produções que vão ser usadas, mas cujas definições podem ser ligeiramente diferentes do usual.

```
sep = r"[,;]"

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

Listagem 5.4: Produções base da gramática

5.1.2.2 Instruções

```
statement <- assignment / voice_declaration / import / for / while / if / return / expression

import <- "import" identifier
      / "import" string_value

assignment <- expression assignment_infix? "=" expression

assignment_infix <- "+" / "-" / "/" / "*" / "|" / "&"

voice_declaration <- ":" identifier "=" voice_declaration_body

voice_declaration_body <- ":" identifier "(" expression ")"
                  / expression

for_variables <- variable ( (',' / ';') variable )*

for_loop_head <- "for" "(" for_variables "in" value_expression ".." value_expression ")"
           / "for" "(" for_variables "in" value_expression ")"

for <- for_loop_head "{" body? "}"

while <- "while" "(" expression ")" "{" body "}"

if <- "if" "(" expression ")" "{" body "}" "else" "{" body "}"
   / "if" "(" expression ")" "{" body "}"

return_statement <- "return" expression?
```

Listagem 5.5: Produções de instruções na gramática

A nível de instruções, a nossa gramática segue o habitual e não carece de muitos comentários.

5.1.2.3 Expressões

As nossas expressões, como já foi dito, estão aqui listadas. O conceito de recursividade à esquerda é ignorado nestes exemplos por questões de simplicidade, e a precedência de operadores binários é denotada apenas informalmente pela ordem das produções, começando pelo operador com menos precedência e subindo até aos operadores com mais precedência.

Os *arrays* e *objetos* são bastante similares à sintaxe usada no *python*, com a distinção de que para evitar ambiguidades com outras produções da gramática, têm de ser precedidos com o símbolo @.

Alguns das outras regras (como notas ou teclados, que são aceites como expressões) serão descritas nas próximas sub-secções.

```
expression <- expression "|" expression
         / expression ("and" / "or") expression
         / expression (">=" / ">" / "==" / "!=" / "<=" / "<") expression
         / expression ("+" / "-") expression
         / expression ("**" / "*" / "/") expression
         / expression expression
         / expression "::" identifier
         / expression "::" "[" expression "]"
         / ( expression / identifier ) "(" parameters ")"
         / "not" expression
         / "(" expression ")"
         / "{" body "}"
         / variable / function_declaration / python_expression
         / keyboard / note / chord / rest / modifier
         / integer / none / bool / string / array / object

variable <- "$" identifier

parameters <- positional_parameters ( parameters_separator named_parameters )?
         / named_parameter?

positional_parameters <- expression ( r"[,;]" expression )*

named_parameters <- named_parameter ( r"[,;]" named_parameter )*

named_parameter <- identifier '=' expression

function_declaration <- "fun" identifier? "(" argument_list? ")" "{" body "}"
                / "fun" identifier? "(" argument_list? ")" "=>" expression

argument_list <- argument ( (',' / ';') argument )*

argument <- ("expr" / "ref" / "in")? "$" identifier ("=" expression)?

python_expression <- "@py" "{" python_expression_body "}"

python_expression_body <- r"[^\}]*"

array <- "@[" "]"
      / "@[" expression ( sep expression )* "]"

object <- "@{" "}"
      / "@{" object_item ( sep object_item )* "}"

object_item <- object_key _ '=' _ expression
```

```
46
47 object_key <- identifier / float / integer / string
```

Listagem 5.6: Produções de expressões na gramática

5.1.2.4 Música

No que toca a música, a sintaxe tenta seguir ao máximo a notação usada pelo projeto **abc**. As suas particularidades também já foram discutidas em bastante detalhe ao longo deste documento, pelo que as produções colocadas aqui não necessitam de muitos comentários.

```
1  note <- note_pitch note_value?
2
3  note_value <- "/" integer
4           / integer "/" integer
5           / integer
6
7  note_pitch <- note_accidental note_pitch_octave
8
9  note_accidental <- "^" / "^^" / "__" / "_" / ""
10
11 note_pitch_octave <- r"[cdefgab]" "'"*
12                 / r"[CDEFGAB]" ","*
13
14 chord <- "[" note_pitch chord_suffix "]" note_value?
15       / "[" note_pitch+ "]" note_value?
16
17 chord_suffix <- 'm7' / 'M7' / 'dom7' / '7' / 'm7b5' / 'dim7' / 'mM7'
18             / '5'
19             / 'M' / 'm' / 'aug' / 'dim' / '+'
20
21 rest <- "r" note_value?
22
23 modifier
24    <- r"[tT]" integer
25     / r"[vV]" integer
26     / r"[iI]" integer
27     / r"[lL]" note_value
28     / r"[sS]" integer "/" integer
29     / r"[sS]" integer
30     / r"[oO]" integer
31     / ":" identifier
```

Listagem 5.7: Produções de elementos musicais na gramática

5.1.2.5 Teclados

As produções para descrever teclados são um ponto fulcral na gramática e na utilização da linguagem em geral. Quisemos que elas permitissem descrever teclados de uma forma concisa, mas ainda assim dando bastante poder ao utilizador na forma como o utiliza.

Com isso em mente, podemos ver que os teclados têm um corpo que difere do corpo geral da linguagem, mas ao mesmo tempo mantém algumas similaridades, nomeadamente, o facto de cada instrução no corpo ser separada por um ponto e vírgula.

No entanto, as instruções no corpo de um teclado são diferentes: a instrução mais básica, denominada por *shortcut*, pode ser imaginada como um par tecla e ação (com mais alguma informação adicional, como por exemplo, argumentos do evento).

Para além disso, também há outras instruções aceites que permitem aos nossos teclados serem gerados dinamicamente: `for`, `while` e `if`. Os cabeçalhos destas instruções são de grosso modo iguais aos das suas versões *general-purpose*, mas nos seus corpos aceitam só instruções de teclados também.

Por fim, existe mais uma instrução válida, *block*, que permite colocar instruções normais e sem restrições, envoltas num par de chavetas. Isto é útil quando precisamos de declarar alguma variável dentro do teclado para evitar, por exemplo, repetir um pedaço de código múltiplas vezes.

```
1  keyboard <- "@keyboard" identifier* group? "{" keyboard_body "}"
2
3  keyboard_body <- keyboard_body_statement ( ";" keyboard_body_statement )* ";"?
4
5  keyboard_body_statement <- keyboard_for
6                     / keyboard_while
7                     / keyboard_if
8                     / keyboard_block
9                     / keyboard_shortcut
10
11 keyboard_for <- for_loop_head "{" keyboard_body "}"
12
13 keyboard_while <- "while" "(" expression ")" "{" keyboard_body "}"
14
15 keyboard_if <- "if" "(" expression ")" "{" keyboard_body "}" "else" "{" keyboard_body "}"
16           / "if" "(" expression ")" "{" keyboard_body "}"
17
18 keyboard_block <- "{" body "}"
19
20 keyboard_shortcut <- keyboard_shortcut_key keyboard_argument_list? ":" expression
21
22 keyboard_shortcut_key <- identifier ("+" identifier)* identifier*
23                   / string_value identifier*
24                   / "[" expression "]" identifier*
25
```

```
26 keyboard_argument_list <- "(" ( variable ( r"[,;]" variable )* )? ")"
```

Listagem 5.8: Produções associadas a teclados na gramática

5.1.3 Reconhecedor Sintático

Para implementar o reconhecedor sintático ou *parser* da linguagem, foi utilizado o módulo *Python* **Arpeggio**, um módulo que implementa um algoritmo de *parsing* PEG descendente recursivo com recurso opcional a *memoization* para melhorar a *performance*.

```
1  main <- body EOF;
2
3  body <- statement ( ";" statement )* _ ";"? _
4     / ""
5     ;
6
7  // Statements
8  statement <- _ ( var_declaration / voice_declaration / function_declaration /
       ↪ for_loop_statement / while_loop_statement / if_statement / expression ) _;
9
10 var_declaration <- "$" namespaced _ "=" _ expression;
11 // ...
```

Listagem 5.9: 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_`).

```
1 def visit_body ( self, node, children ): ...
2
3 def visit_comment ( self, node, children ): ...
4
5 def visit_statement ( self, node, children ): ...
6
7 def visit_var_declaration ( self, node, children ): ...
```

Listagem 5.10: Métodos responsáveis por criarem a AST

5.1.4 Conversão de PEG para LALR

Uma vez estando a gramática relativamente estabilizada, foi medida a sua *performance*, marcando o tempo que ela demorava a reconhecer os ficheiros em formato Musikla da biblioteca *standard* (que são carregados sempre que algum *script* Musikla é executado). Os valores obtidos rondavam em média os

600 milissegundos. Apesar de ser aceitável atualmente, não inspiravam confiança caso o tamanho da biblioteca crescesse no futuro (o que irá certamente acontecer, e que causará esse tempo aumentar). Principalmente por essa razão, decidimos converter a gramática PEG para uma gramática Look Ahead Left Right (LALR) (que pode ser visualizada no apêndice B).

Para esse efeito escolhemos o gerador de reconhecedores sintáticos **lark**[1]. Esta ferramenta tem várias funcionalidades que captaram o nosso interesse:

Gerar Múltiplos Reconhecedores A biblioteca permite escolher entre dois reconhecedores para a mesma gramática, cada um oferecendo vantagens e desvantagens específicas:

Earley Permite reconhecer qualquer gramática livre de contexto, até capaz de reconhecer gramáticas ambíguas (e deixar o utilizador resolver as ambiguidades, ou receber todas as possíveis interpretações de um dado *input* perante uma gramática ambígua).

LALR Menos poderosa mas bastante mais eficiente em memória e velocidade, reconhecendo o *input* em tempo linear $O(n)$.

Sintaxe Rica Inspirada pela notação EBNF[2] com funcionalidades extra que fazem escrever, ou no nosso caso portar a gramática, uma tarefa bastante simples.

Reconhecedor Léxico Incremental Em vez de reconhecer os *tokens* de uma vez e os passar depois ao reconhecedor sintático, permite que os *tokens* vão sendo reconhecidos dependendo do contexto em que o reconhecedor sintático se encontra. Assim, regras terminais que podiam ser ambíguas e reconhecer o mesmo *input*, são facilmente desambiguadas com base no sítio em que estão.

O resultado final foi uma gramática bastante similar à anterior, mas cujo reconhecedor sintático é aproximadamente 10 vezes mais rápido (demorando agora cerca de 50 milissegundos para reconhecer a biblioteca *standard*). As maiores diferenças da gramática passam pela forma como os *whitespaces* são tratados. Estes são ignorados pelo reconhecedor sintático, o que não faz mal pois na linguagem Musikla apenas a chamada de funções é *whitespace-aware*. Parênteses curvos colados a algum *token* são uma chamada de função, enquanto que separados são um agrupamento de expressões. Para fazermos esta distinção, alguns*tokens* que precedem a chamada de uma função no reconhecedor léxico têm duas versões: uma que inclui o parênteses de abertura no próprio *token*, sem espaços antes, e que permite identificar a chamada de uma função.

5.1.5 Tooling

Para além das gramáticas e do *parser* da linguagem, que são indispensáveis para o funcionamento do projeto, foram também implementados algumas funcionalidades adicionais com o intuito de facilitar a utilização da nossa linguagem, que só vamos cobrir brevemente.

[1] Lark Website https://lark-parser.readthedocs.io/en/latest/

[2] Extended Backus–Naur form https://en.wikipedia.org/wiki/Extended_Backus-Naur_form

5.1.5.1 Syntax Highlighting

Para oferecer *syntax highlighting* para a nossa linguagem, que a nosso ver torna a experiência de programar com a mesma muito mais agradável, utilizamos uma linguagem chamada **Iro**[3]. Esta linguagem é uma DSL que permite descrever a sintaxe através de expressões regulares, e associar estilos a cada uma.

```
:pattern {
   regex \= (=>|<=|>=|==|!=|[=\*\|\+\-\/]|>=)
   styles [] = .operator;
}

: pattern {
   regex \= (#.*)
   styles [] = .comment;
}
```

Listagem 5.11: Pequeno excerto da definição da linguagem escrito em Iro

A sua grande vantagem é permitir, a partir de uma única definição, gerar versões em formatos específicos suportados por vários editores, como por exemplo o *TextMate*, *Visual Studio Code*, *Atom*, *Sublime 3*, ou até convenientemente para o formato suportado pelo popular módulo *Python* **Pygments**. Usamos este módulo para colorir os exemplos de código na nossa documentação, como iremos ver na secção seguinte.

[3] https://eeyo.io/iro/

```
42  $toggle_sustain = fun ( ref $enabled ) => {
43      if ( $enabled ) {
44          cc( 64; 0 );
45      } else {
46          cc( 64; 127 );
47      };
48
49      $enabled = not $enabled;
50  };
51
52  fun keyboard () {
53      $sustained = true;
54
55      @keyboard { q toggle: accomp() | melody(); };
56
57      @keyboard hold extend {
58          a: [^Cm];
59          s: [BM];
60          d: [AM];
61          f: [EM];
62          g: [^Fm];
63      };
64
```

Figura 5.1: *Syntax highlighting* da linguagem MusiKLa no editor Visual Studio Code

5.1.5.2 Documentação

Também foi escrita documentação para a linguagem disponível online[4], que inclui informação sobre a sintaxe, exemplos de *scripts* funcionais escritos em *musikla*, e uma API com a descrição das várias funções disponibilizadas, as suas assinaturas e alguns exemplos de utilização.

A documentação foi escrita em *Markdown*, e convertida para HTML usando o projeto **mkdocs**[5]. Os exemplos de código disponibilizados na documentação são coloridos pelo módulo **Pygments**, usando a definição de linguagem que descrevemos na secção anterior.

5.1.5.3 Relatórios de Erros

A aplicação de linha de comandos que interpreta a linguagem também dispõe de um componente responsável por, quando ocorre algum erro relacionado com a linguagem, imprimir para o terminal o sítio onde o erro ocorreu, bem como a mensagem a descrever o erro.

Existem dois tipos de erros possíveis: erros de sintaxe, que ocorrem durante a fase de *parse* quando algum carácter está no sítio errado; e erros de execução, que ocorrem quando a linguagem está a executar e podem ter causas variadas.

Para ambos os erros, o componente de relatório precisa apenas dos seguintes campos:

Ficheiro Qual o caminho do ficheiro onde ocorreu o erro (opcional).

[4] https://pedromsilvapt.github.io/miei-dissertation/
[5] https://www.mkdocs.org/

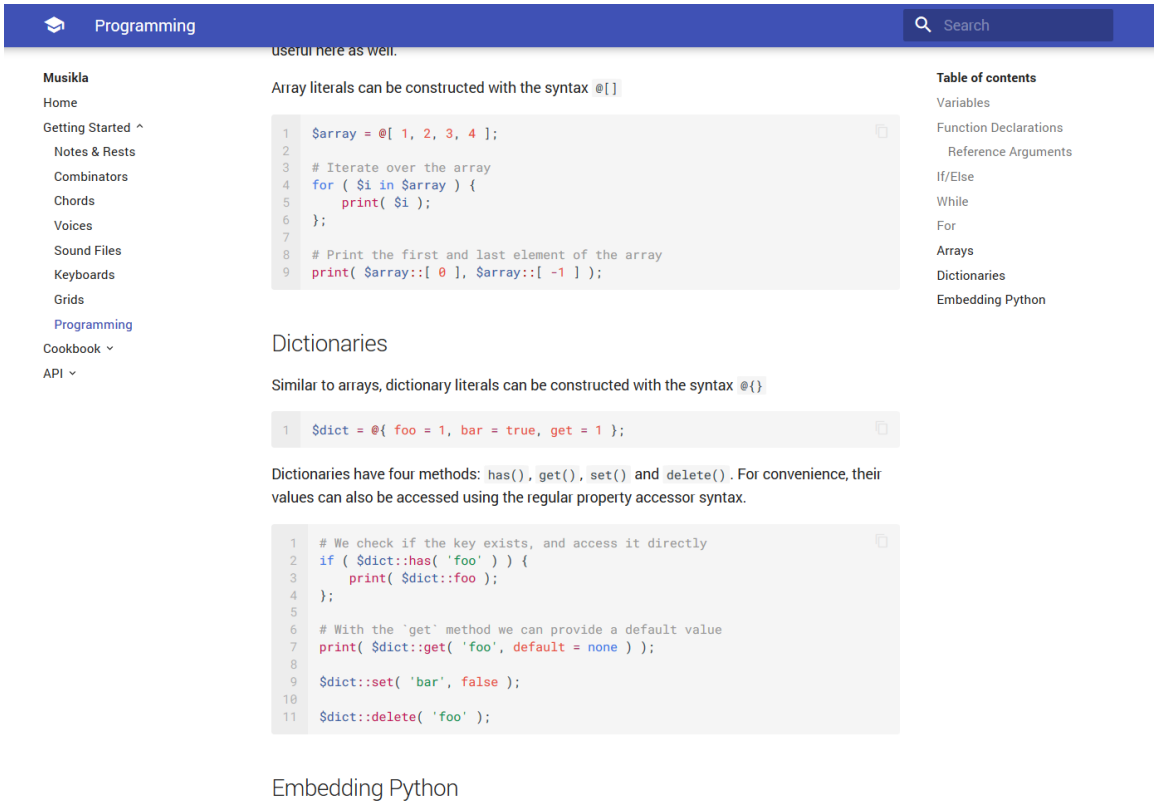


Figura 5.2: Captura de ecrã da página de documentação da linguagem

```
ParseError in .\musikla\examples\fuga.mkl:7:32
04 |  cc( 64, 0 );
05 |
06 |  fun fugue ( $subj, $resp ) =>
07 |      ( $subj $resp | r ** $subj ; ( $subj + 7 ) );
08 |
09 |  # Flat: A B e
10 |  $subj = r c/2 B/2 c G _A c/2 B/2 c d

Message Unexpected token Token(SEMICOLON, ';') at line 7, column 32.
```

Figura 5.3: Relatório de erro de *parse*

Código O conteúdo do ficheiro que originou o erro. Este campo pode parecer redundante, tendo nós o caminho do ficheiro. Seria de esperar que podíamos simplesmente ler o ficheiro e obter o seu conteúdo. No entanto, desde o momento que o ficheiro foi lido inicialmente, e o momento em que ocorreu o erro, o seu conteúdo pode ter mudado. Por esta razão, é importante manter em memória o código dos ficheiros carregados para assegurar que os erros são mostrados corretamente.

Posição Início e Fim Dois inteiros indicando o inicio e fim da secção contígua no ficheiro onde o erro deve ser marcado a vermelho. Podem cobrir mais do que uma linha.

Mensagem Finalmente, qual a mensagem de erro a mostrar ao utilizador.

Erros de Sintaxe

Estes erros são fáceis de reportar. Sempre que há algum erro de sintaxe, o *parser* emite uma exceção já com a informação da mensagem de erro, bem como a posição onde o erro ocorreu. Na nossa aplicação, só temos de tratar essa exceção, adicionando-lhe o caminho e o conteúdo do ficheiro a que estávamos a fazer *parse*, e já temos toda a informação disponível para mostrar o erro.

Erros de Avaliação

Este tipo de erros pode ocorrer em qualquer lado. Por essa razão, antes de fazer-mos *parse* de qualquer ficheiro, associamos o ficheiro (e o seu conteúdo) a um inteiro único por execução.

Depois durante a fase de *parsing*, associamos a cada nó da AST um triplo com três inteiros, representando o ficheiro, posição de início, e posição de fim a que esse nó corresponde.

Desta forma, no método `Node.eval()` herdado por todos os nós da AST, colocamos um `try ... except ...` à volta da chamada de `Node.__eval__()` (função que contém o código específico de execução de cada nó). Desta forma, conseguimos adicionar a qualquer exceção que ocorra durante a execução, o triplo que o nó tem com a informação sobre o ficheiro e a sua posição no mesmo.

Esta informação é depois utilizada para preencher os campos e mostrar o relatório de erro.

5.2 Semântica Dinâmica

As linguagens compiladas usualmente recorrem à compilação Ahead Of Time (AOT), onde o código é transformado em código máquina antes de ser executado (durante a fase de compilação). Esta é a solução 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 para melhor tirar partido das vantagens de *performance*.

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 na dificuldade de implementação que são 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 referidas 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 *tree walk*).

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 otimizado 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 geralmente 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 substituída 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 uma simples **Bytecode VM**. Acabamos por escolher a primeira opção pelas seguintes razões:

- A geração de eventos músicais é relativamente barata em termos computacionais (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 mais optimizadas e desenvolvidas em linguagens de baixo nível.
- A possibilidade de correr código *Python* em qualquer parte da nossa linguagem já fornece um bom meio termo para quando é necessária mais alguma *performance* em caminhos de execução críticos (*hot paths*), 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 ordens de magnitude superior do que seria possível de outra forma, 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 musicais 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 musicais 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 musicais 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 musicais 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**.

Apesar de estarmos a lidar com notas e eventos musicais, fundamentalmente estamos a criar ramos paralelos de execução, e queremos no final aguardar o seu resultado e uníficá-lo com o ramo original. Se substituirmos ramo por *thread* para o caso da programação, ou *contexto* para o nosso caso, vemos a equivalência entre os conceitos.

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 objeto *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 visíveis 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 impossível 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 Musikla 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 Musicais

A sintaxe suporta os operadores aritméticos e de comparação usuais nas linguagens de programação. De modo geral, a sua sintaxe e semântica na nossa linguagem, o que seria expectável para cada um deles, pelo que não vale enumerá-los a todos nesta secção. No entanto, existem alguns operadores menos convencionais, ou até *overloads* de operadores convencionais (como o operador de multiplicação e adição) que funcionam de forma diferente quando empregues junto a valores do tipo musical. São esses os casos que iremos abordar nas seguintes sub-secções.

Um aspeto importante a ter em conta é que estes operadores não estão restritos gramaticalmente para aceitar apenas expressões musicais, mas aceita sim qualquer tipo de expressão. O seu tipo (e portanto a sua semântica) são apenas decididos em *runtime* (imitando o modo de funcionamento do *overloading* de operadores em *Python*). Isto é necessário porque uma variável pode conter tanto um número como uma expressão musical, e o seu verdadeiro valor apenas é conhecido em execução.

5.2.4.1 Sequencial/Concatenação

Ao longo deste documento já vimos várias instâncias de como declarar uma nota ou um acorde. O ato de concatenar esses eventos (ou seja, reproduzi-los em sequência) é tão fundamental para a linguagem que também é algo que já vimos inúmeras vezes nos exemplos sem pensar muito nisso. Uma das razões para isso deve-se à sua sintaxe (ou falta dela, na verdade). O ato de concatenar eventos musicais é tão simples como escrevê-los uns à frente dos outros, sem nenhum tipo de separador especial (para além dos opcionais espaços em branco). O evento ficam com o seu tempo inicial sempre igual ao tempo final do evento anterior.

```
1 S4/4 T74 L/8 V90;
2 A, E A B ^c B A E D ^F ^c e ^c A3;
```

Listagem 5.12: Excerto da música *Wet Hands* de C418

Também é importante salientar o facto de que as listas de instruções (*statements*) seguem a mesma semântica do operador de concatenação. Cada linha só vai tocar as suas notas quando as linhas anteriores

[6] Renderizado com a biblioteca $ABC_UI, com alguns ajustes manuais feitos para melhorar a clareza.
[7] Concatenação https://drive.google.com/open?id=1TP4lcul81s8iMCUFmD3HKnSpeftCzKT0

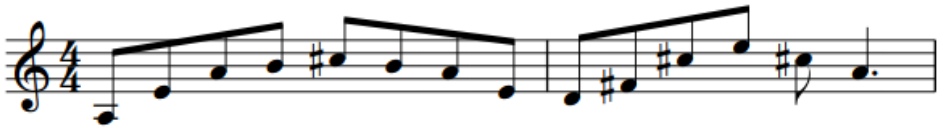


Figura 5.4: Pauta musical gerada pela linguagem[6], versão áudio disponível aqui[7].

tiverem acabado. Neste exemplo portanto, ter tudo na mesma linha ou separado em duas (ou mais) tem o mesmo resultado, variando apenas a legibilidade.

5.2.4.2 Repetição

O operador de repetição é bastante similar ao operador de concatenação. E faz sentido, uma vez que repetir uma expressão musical N vezes pode ser pensado como um caso particular da concatenação onde existem N operandos, todos iguais, uns em frente aos outros.

```
1 I1 S6/8 T140 L/8 V90;
2 A*11 G F*12
```

Listagem 5.13: Excerto do começo do tema principal de Westworld, por Ramin Djawadi

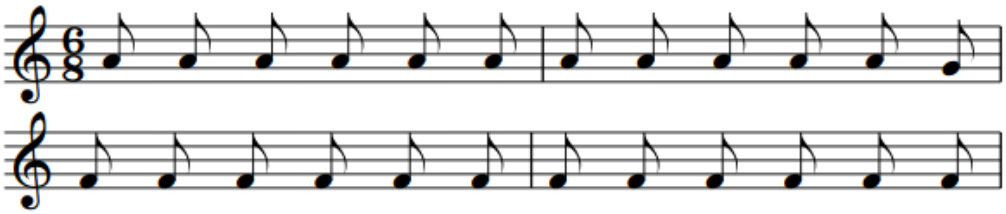


Figura 5.5: Pauta musical gerada pela linguagem, versão áudio disponível aqui[8].

Este operador, apesar de bastante simples, é também bastante útil para repetir pequenos acompanhamentos (mesmo notas singulares) muitas vezes, ou até para repetir poucas vezes acompanhamentos mais complexos. Evitando repetir os acompanhamentos no código, é possível efetuar alterações em apenas um sítio e ver essas alterações repetirem-se ao longo de toda a estrutura musical.

5.2.4.3 Paralelo

O operador paralelo permite reproduzir múltiplas sequências de eventos musicais em paralelo. Podemos dizer que conceptualmente, o operador paralelo é uma função que recebe uma lista de sequências musicais, e retorna uma única sequência com todas as sequencias combinadas. No entanto é importante que lembrar que o operador tem de respeitar os princípios de ***lazyness*** (não pedir mais eventos de cada vez do que o que os que são estritamente necessários) e **ordem** (a sequência de eventos resultantes tem de estar estritamente ordenada pelo tempo de início de cada evento).

[8] Repetição https://drive.google.com/open?id=1IIm8PQkLsNFMK9MNSVubJSG6SP6KwhPL

```
1 T120 V70 L1;
2 r/4 ^g/4 ^g/4 ^g/4 ^f/2 e/8 ^d3/8 ^c2 | [^Cm] [BM] [AM] [BM]
```

Listagem 5.14: Excerto da música *Soft to Be Strong* de Marina

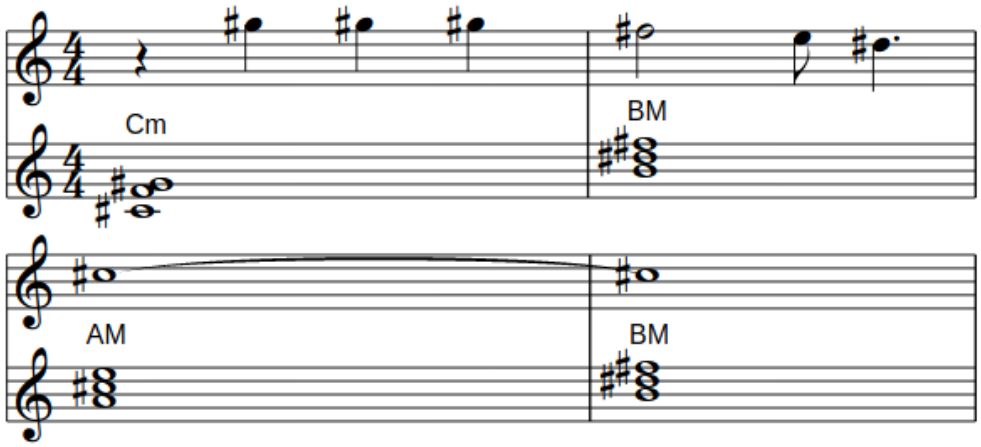


Figura 5.6: Pauta musical gerada pela linguagem, versão áudio disponível aqui[9].

A implementação do operador paralelo depende de um simples algoritmo de *merge sorted* (não relacionado com o conhecido algoritmo de ordenação de *arrays merge sort* inventado por John von Neumann).

A função *merge sorted* recebe N operandos e cria um *buffer* também de tamanho N. Para cada operando, ela cria um *fork* do seu contexto, para que cada operando possa executar concorrentemente e assim modificar apenas o seu próprio contexto. Depois, ainda durante a inicialização do algoritmo, a função pede a cada operando um evento (que são guardados nas respetivas posições do *buffer*).

Após o *buffer* estar preenchido pela primeira vez com um evento por operando, o algoritmo procura o evento mais recente (o evento com *timestamp* menor) guardado no *buffer*. Vamos assumir que esse evento se encontra guardado na posição K do *buffer*, sendo $k < N$. O método emite esse evento `buffer[K]` para a sequência musical de saída, e de seguida pede ao operando na posição K o seu próximo evento, usando-o para substituir o que estava anteriormente no *buffer* nessa mesma posição (guardando o valor `null` caso o operando K já tenha chegado ao fim). O algoritmo repete depois este passo até todos os operandos terem chegado ao fim.

5.2.4.4 Transposição

As operações `Music + Integer` e `Music - Integer` correspondem à transposição de notas (e acordes). Podem ser aplicados a notas singulares ou expressões musicais complexas, o que resulta na transposição a ser aplicada a cada uma das notas da expressão musical.

```
1 L/8;
```

[9] Paralelo https://drive.google.com/open?id=1ENTm3hZonYHyQIOgRZ8TQ1Qz-AfRLt2I

```
(A B c) (A B c) + 12;
```

Listagem 5.15: Exemplo de transposição de um acompanhamento com três notas

Figura 5.7: Pauta musical gerada pela linguagem, versão áudio dísponível aqui[10].

Neste simples exemplo listado em 5.15 e ilustrado em forma de pauta musical na Figura 5.7, podemos ver claramente que a música é composta por dois triplos de notas, ambos com o mesmo padrão, sendo a única diferença que o segundo é tocado uma oitava acima (12 semitons).

5.2.4.5 Arpeggio

A operação `Chord * Music` [11] permite aplicar um **Arpeggio**. Esta operação consiste em pegar nas notas presentes no primeiro operador e colocá-las numa lista de $0..N$ não inclusivo (sendo N o número de notas no acorde). De seguida é avaliada a segunda expressão musical, mantendo os seus tempos e posições, mas substituindo as notas pelas pertencentes ao acorde. O modo de substituição usa a seguinte escala: `C D E F G A B`. Isto é, assume a nota `C` como base (a que é substituída pela primeira nota do acorde), a nota `D` sendo substituída pela segunda nota do acorde, e por aí fora.

```
S4/4 L/8 T85;

$p = C D E D2 E C2;

[CeG]*$p [Ac'e]*$p [Fac]*$p [CeG]*$p;
```

Listagem 5.16: Três acordes diferentes arpegiados com o mesmo padrão

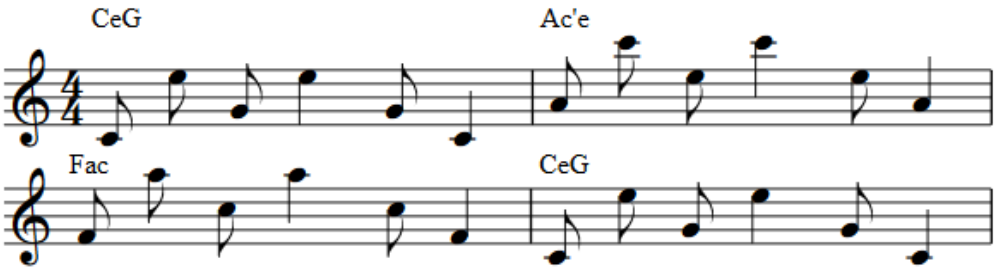


Figura 5.8: Pauta musical gerada pela linguagem, versão aúdio dísponível aqui[12].

No exemplo da listagem 5.16 recorremos ao uso de uma variável para guardar o padrão da nossa melodia. De seguida usamos o operador de *arpeggio* em cada um dos acordes, sempre com o mesmo

[10] Transposição https://drive.google.com/file/d/1xVey6on6Q-sWjNgo__LIb3QgOF1-_axa
[11] Na implementação atual, o operador aceita não só acordes como primeiro operando, mas qualquer expressão musical
[12] *Arpeggio* https://drive.google.com/file/d/1E6ayR6gG2CfsikWFHijnEIui0rR1cAL7

padrão. Esta funcionalidade permite definir uma melodia e testar rapidamente múltiplas variações da mesma sem ter de alterar manualmente a melodia de cada vez.

Atualmente a definição do padrão recorre à sintaxe das músicas usuais, por conveniência. No futuro, seria interessante implementar uma sintaxe específica para a definição de *arpeggios*, usando por exemplo numeração romana. Dessa forma, o padrão do exemplo na listagem 5.16 ficaria algo como `I II III II2 III I2`. Outra possibilidade seria a de usar a numeração decimal, com alguma marcação para distinguir dos números inteiros, tal como `1º 2º 3º 2º2 3º 1º2`.

5.2.4.6 Redimensionamento

O operador de **Redimensionamento**, denotado pela sintaxe `Music ** Integer`, `Music ** Fraction`, ou `Music ** Music`, pode talvez ser o menos intuitivo, mas é algo bastante útil. Ele permite mudar o tamanho de uma expressão arbitrária já declarada, para combinar com outra duração expressa num inteiro, fração, ou inferida através da duração de uma segunda expressão musical. O seu valor de retorno é então o primeiro operando, mas com os tempos proporcionalmente esticados (ou encolhidos) de modo a, no total, ficarem com a nova duração escolhida.

```
1 S2/4 L/8 T90;
2
3 $chords = V90 [CeG] [Ac'e] [Fac] [CeG];
4 $melody = CGde2gc2 CGdr5;
5
6 $chords ** $melody | $melody;
```

Listagem 5.17: Redimensionamento da duração de uma expressão musical

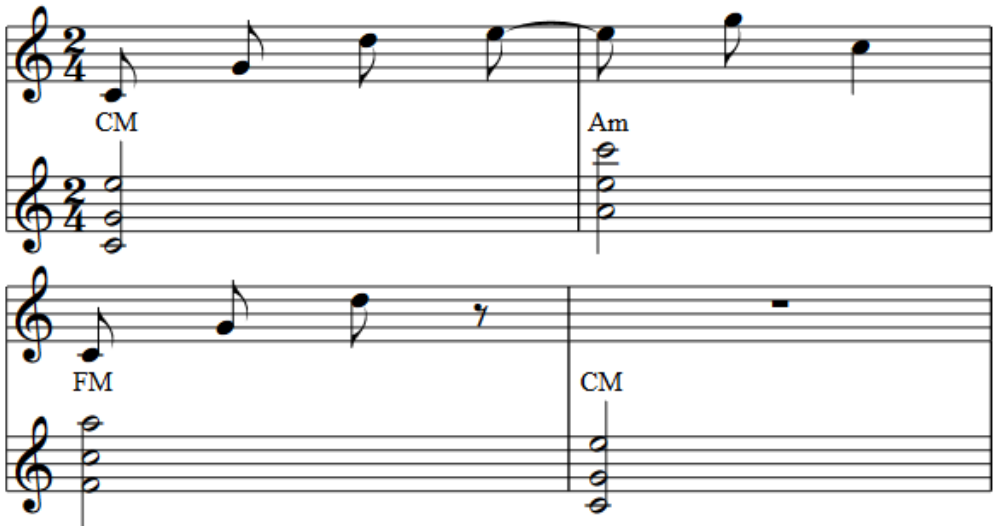


Figura 5.9: Pauta musical gerada pela linguagem, versão áudio dísponível aqui[13].

[13] Redimensionamento https://drive.google.com/file/d/1aVVXGDVQEAmHI-jadSnZ6OqIGmgqhgxR

No exemplo presente na listagem 5.17, temos duas variáveis com durações diferentes. A primeira, **`$chords`**, tem a duração total de $\frac{1}{2}$ ($4 \times \frac{1}{8}$). Por sua vez, a segunda variável, **`$melody`**, tem a duração de **2** ($16 \times \frac{1}{8}$). As duas durações podiam ser ajustadas manualmente (bastando multiplicar a duração de cada acorde por 4), mas isso implicava que qualquer mudança na melodia exigia recalcular as proporções manualmente. E também nem todas as situações apresentam contas tão simples como $2 \div \frac{1}{2}$. Deste modo, os cálculos são feitos automaticamente pela aplicação e estão sempre atualizadas.

5.3 Biblioteca Standard

A biblioteca *standard* que é incluída com o interpretador Musikla foi também um dos grandes desafios do desenvolvimento deste projeto. Obviamente, o seu desenvolvimento ocorreu de uma forma faseada, ao mesmo tempo que a linguagem e o interpretador eram desenvolvidos. Isso significa que desenvolver a biblioteca não é apenas a tarefa de adicionar novas funcionalidades, mas também de reescrever ou adaptar funcionalidades existentes que tenham sido adicionadas numa fase inicial do desenvolvimento da linguagem. Por essa razão as suas interfaces públicas podiam estar desenhadas à volta de uma linguagem que tinha menos funcionalidades que a versão atual, sejam elas objetos, listas ou passagem de argumentos com nome.

No diagrama abaixo podemos ver uma lista dos módulos que são disponibilizados pela linguagem. As funções e classes listadas abaixo são apenas um pequeno exemplo do que é implementado na biblioteca. A documentação mais completa pode ser encontrada online.

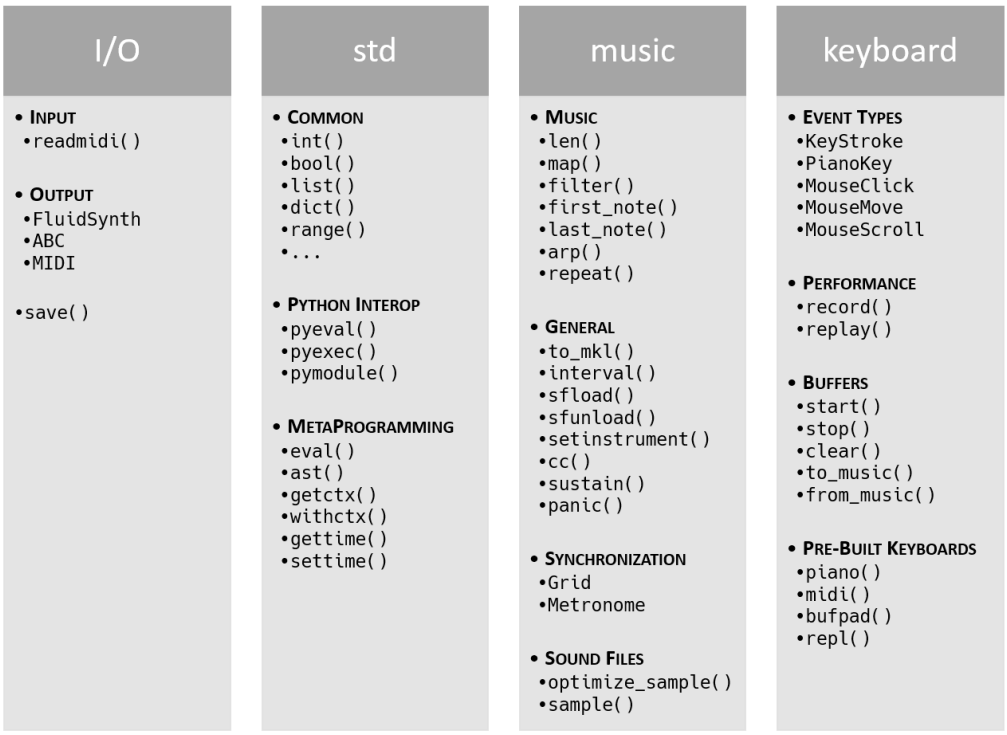


Figura 5.10: Organização dos módulos *standard* do projeto

Todos estes módulos, por serem tão comuns e essenciais, são sempre carregados automaticamente antes de executar um *script* Musikla, pelo que não há necessidade de os importar manualmente.

5.3.1 Inputs & Outputs

Durante a execução, os eventos musicais 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 sintaxe 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 musicais. De facto, existem já vários formatos de descrição de música, e a capacidade de os poder ler e traduzir para objetos musicais dentro da nossa linguagem, indistinguíveis daqueles que são criados pela sintaxe de literais, traz inúmeras vantagens e abre a possibilidade de este projeto se tornar como um canivete suíço da notação musical, capaz de ler, transformar e depois também escrever em diversos formatos diferentes.

Para além de ser capaz de ler (**input**) eventos musicais 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 musicais em diversos formatos.

O verdadeiro objetivo da linguagem é que seja extensível, de modo a que qualquer pessoa capaz de programar *Python* possa conectar 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

A biblioteca **FluidSynth** serve 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*(Whitehead, 2019) 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 necessitávamos, 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 musicais) 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. Uma das vantagens é o facto de o FluidSynth já disponibilizar uma interface segura de acesso entre *threads*(Henningsson & Team, 2011).

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 musicais, reproduzir ficheiros musicais. 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 musical 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 é pedido para reproduzir o mesmo ficheiro várias vezes, a mesma nota do mesmo instrumento é utilizada, evitando estar sempre a ler os conteúdos do ficheiro. Este evento musical 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 musicais 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

Uma funcionalidade extremamente importante e necessária para 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 musicais 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 musicais de um ficheiro ou porta MIDI, retornando um objeto do tipo **Music**. A função aceita os seguintes parâmetros (sendo os parâmetros de ficheiro ou de porta mutuamente exclusivos):

file *(Opcional)* Quando presente, lê os eventos musicais 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. É 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 atribuídos 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

Outro dos formatos de saída suportados é o formato **abc notation**. A sintaxe das notas e acordes da nossa linguagem é fortemente inspirada nesta notação, pelo que não seria estranho pensar que escrever para este formato não seria nada de especial. E em parte é verdade, na medida que gerar o texto com o formato certo não é complicado.

A maior diferença é, no entanto, o facto de que o formato *abc* é estático, e a sua estrutura foi desenhada para imitar de certa forma a notação das partituras musicais. Tudo isto envolve conceitos como agrupar as notas por vozes, pautas, compassos, claves, entre outros. De certa maneira, o modelo de dados que a linguagem usa para representar as notas em memória mapeia-se de forma muito mais simples para o formato *MIDI* do que para o *abc*.

Para colmatar este facto, os eventos, antes de serem convertidos para o formato *abc*, são introduzidos numa *pipeline* que executa uma série de passos de modo a tornar a conversão mais simples. De grosso modo, os passos pela ordem que são executados, são os seguintes:

1. Em primeiro lugar o sistema tenta converter quaisquer notas que possam estar a ser emitidas como eventos *on/off* separados. Isto acontece quando, por exemplo, o teclado é usado, o que pode implicar começar a tocar as notas sem saber quando o utilizador as vai parar. Os pares de eventos são então convertidos num só evento com a informação do início e da duração da nota.

2. Em segundo lugar, o sistema tenta identificar as notas paralelas que possam estar a ser emitidas e que possam pertencer a um acorde. Isto acontece mais uma vez quando, ao invés de usar a sintaxe de acordes presente na linguagem, o usa o teclado para gerar o acorde através de notas/teclas diferentes ativadas ao mesmo tempo.

3. Em terceiro lugar, as notas que pertençam à mesma voz mas que tenham *overlap* na sua duração, são colocadas em linhas diferentes. Mais uma vez, isto acontece mais frequentemente quando o teclado é usado, onde o utilizador pode estar a tocar notas concorrentemente da forma que quiser.

4. Em quarto lugar, são inferidos eventos pausa quando duas notas consecutivas pertencentes à mesma voz têm um intervalo. Mais uma vez, isto acontece mais frequentemente quando o utilizador toca com o teclado. Mas neste caso pode também acontecer quando as notas são declaradas pela

sintaxe da linguagem, como por exemplo na seguinte situação `A ( B | C )`, onde as notas A e B ficam na mesma pauta. A nota C, no entanto, como se sobrepõe com a nota B, pelo passo anterior, seria movida para a sua pauta. No entanto, esta pauta não teria nada ao início, e por isso precisa de uma pausa. Seria conceptualmente equivalente ao resultado de `(A B) | (r C)`.

5. Finalmente em quinto lugar, são introduzidos eventos especiais na sequência musical para identificar, para cada voz em separado, quando devem ser introduzidos novos compassos, ou novas pautas na partitura. Do mesmo modo, isto implica que notas e acordes cuja duração não caiba num compasso, sejam divididos quantas vezes forem necessários, e conectados através de ligaduras.

Estas transformações são reutilizáveis (não estão fortemente acopladas ao formato *abc*) e são implementadas utilizando o conceito de transformadores, que iremos abordar mais à frente neste documento.

Muitas destas transformações são, no entanto, imperfeitas por natureza: a deteção de acordes a partir de notas paralelas individuais, por exemplo, nunca pode ter 100% de certeza se as notas que agrupou são realmente um acorde. É possível que o músico tivesse a intenção de que as várias notas pertencessem a vozes diferentes, por exemplo.

Para além disto, a notação musical é tão complexa que apenas uma pequena parte é suportada pela nossa linguagem. Deste modo, é preciso ter em mente que apesar de a linguagem ser capaz de gerar ficheiros *abc* válidos, e com a sua estrutura maioritariamente correta de acordo com o que o utilizador tivesse em mente, serão muitas vezes necessários pequenos ajustes manuais para personalizar os resultados aos gostos de cada um. Ainda assim, o facto de não ser necessário escrever o ficheiro todo manualmente é por si só uma vantagem enorme.

Isto porque o ficheiro *abc* não serve só como um formato de texto. Existem várias ferramentas para o converter para outros formatos, como *PDF*. Por exemplo, o projeto **$ABC_UI**[14] é utilizado pela nossa linguagem para, em conjunto com o exportador *abc*, gerar páginas HTML com uma renderização da pauta usando Scalable Vector Graphics (SVG).

Desta forma, podemos concluir que o formato *abc* serve como uma espécie de ponte *standard*, e nos permite apanhar boleia para depois suportar inúmeros outros formatos que existam muito mais facilmente.

5.3.2 Ficheiros de Som

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

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

[14] http://dev.music.free.fr/web-demo/\protect\TU\textdollarABC_UI.html

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()`.

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

Listagem 5.18: 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, adotamos 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*

Tabela 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.

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

Listagem 5.19: 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 Grelhas

A linguagem tem dois modos de produção de música principais: expressões musicais descritas no código, e a utilização de teclados em tempo real. Em termos de propriedades temporais, de modo geral, as expressões musicais são geradas com tempos precisos. Por outro lado, como é natural, músicas criadas através dos teclados musicais vão ter tempos imperfeitos. Quando utilizadas de modo separado, as diferenças muitas vezes são tão subtis que não se notam. Mas quando queremos tocar uma expressão de música pré-definida em código ao mesmo tempo que se utiliza um teclado, as diferenças e imperfeições podem-se tornar mais óbvias (Frühauf, Kopiez & Platz, 2013).

A operação conhecida como *Quantization* [15] é possível na nossa linguagem através da utilização de **Grelhas**. Uma vez que o objetivo das grelhas é poderem ser utilizadas em tempo real à medida que as notas vão sendo tocadas, a forma de funcionamento escolhido para as grelhas é bastante simples e determinística, dependendo apenas do tempo (*timestamp*) e tipo do evento que queremos alinhar.

Uma grelha é composta por um número infinito de células, todas do mesmo tamanho, que se estendem ao longo do eixo temporal. Quando um evento musical é tocado, esse evento vai calhar algures dentro de uma dessas células. A responsabilidade da grelha passa por determinar se o evento deve ser movido no tempo, bem como para onde deve ser movido. Para isso, devemos compreender como cada célula da grelha é dividida.

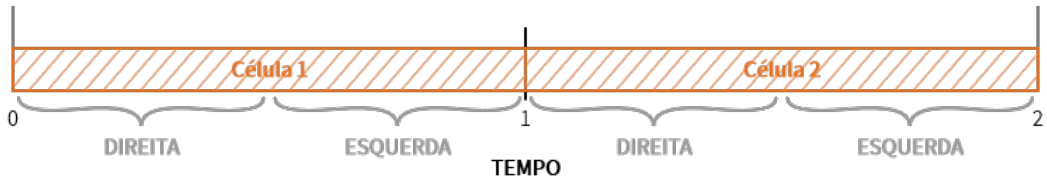


Figura 5.11: Representação de duas células de uma grelha

Na figura 5.11 podemos ver representadas duas células de uma grelha, cada uma com tamanho de 1 segundo. Também podemos ver que cada célula está dividida em duas partes, direita e esquerda, e a ordem dessas partes pode parecer engano, ou que estão trocadas. Mas estão corretas, pois na verdade cada uma dessas partes é relativa não à célula, mas ao separador das células.

Olhemos por exemplo para o divisor central, situado em cima da marca do 1 segundo. A parte à sua esquerda refere-se à esquerda do separador, e a parte à direita referem-se à direita do separador. Desta forma, faz mais sentido a ordem das partes.

Por predefinição, ambas as partes têm o tamanho igual a metade do tamanho da célula (ou seja, neste caso, tanto a parte da esquerda e da direita têm o tamanho de 0.5 segundos). No entanto, é possível customizar o tamanho das duas de forma igual, ou cada uma em particular, através das propriedades **forgiveness** e **range**.

```
1 $grid = Grid( 1,
```

[15] https://en.wikipedia.org/wiki/Quantization_(music)

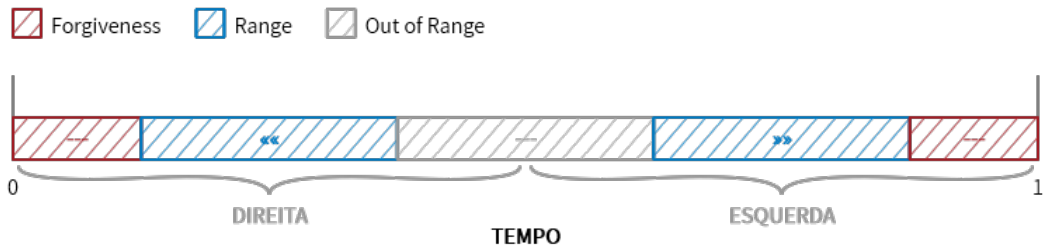


Figura 5.12: As várias áreas configuráveis de uma grelha

```
    forgiveness = 125, # 1/8 of a second
    range = 375 # 3/8 of a second
);
```

Listagem 5.20: Código de definição da grelha representada na figura 5.12

Não nos esqueçamos que o objetivo da grelha é alinhar eventos, movendo os eventos ao longo do eixo do tempo para junto dos separadores das células. A propriedade *range* tem como valor predefinido metade do tamanho da célula. Isso significa que tanto o **range_left** como o **range_right** cobrem de modos igual toda a célula, e dessa os eventos são movidos para o separador da grelha que estiver mais próximo. O valor predefinido da propriedade *forgiveness*, por outro lado, é zero, o que significa que mesmo os eventos que se encontrem já muito perto dos separadores são movidos.

Na listagem 5.20 podemos ver a definição de uma grelha simétrica (as propriedades *left* e *right* são iguais). O valor de *forgiveness* implica que eventos que estejam a mais de 372 milissegundos de distância de qualquer um dos separadores são ignorados pela grelha, isto é, o seu tempo não se mexe. O valor de *forgiveness*, o que significa que eventos que estejam a menos de 125 milissegundos de um dos separadores são também ignorados pela grelha. Neste exemplo, são apenas movidos os eventos que estejam entre 125 e 375 milissegundos de distância de um dos separadores.

Note-se que cada propriedade poderia ser costumizada para a esquerda e para a direita. Isto é, o exemplo acima podia ser reescrito da seguinte forma:

```
$grid = Grid( 1,
    forgiveness_left = 125, # 1/8 of a second
    forgiveness_right = 125, # 1/8 of a second
    range_left = 375, # 3/8 of a second
    range_right = 375 # 3/8 of a second
);
```

Listagem 5.21: Código alternativo de definição da grelha representada na figura 5.12, com as propriedades *left* e *right*

Se passarmos das propriedades de configuração para a grelha resultante, na verdade cada célula passa a ter dois **effective ranges**, duas áreas (uma para a esquerda e outra para a direita) onde os

eventos que aí calharem são movidos. Os eventos fora dessas áreas são ignorados.

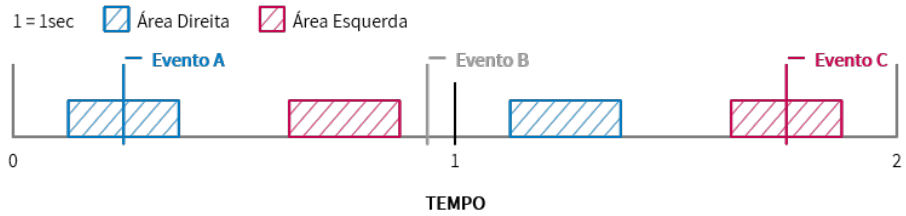


Figura 5.13: Áreas em que a grelha irá mover os eventos, e qual o separador a que pertencem.

Tomemos como exemplo a seguinte situação com três eventos e a grelha definida anteriormente: o evento A, como calha dentro da área azul (devido à propriedade **range_right**), vai ser relocado para o tempo zero. O evento B no entanto, como não calha em nenhuma área (devido devido a propriedade **forgiveness_left**. Finalmente, o evento C como calha na área vermelha (devido à propriedade **range_left** é relocado para o tempo 2.

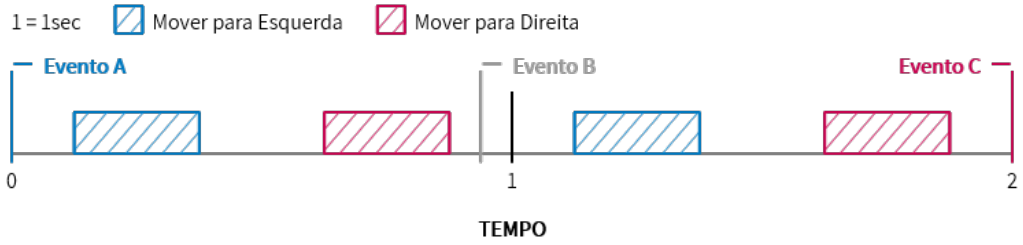


Figura 5.14: Posição dos eventos após o alinhamento com a grelha ter sido aplicado.

O modo de funcionamento das grelhas pode parecer à primeira vista um desnecessariamente complexo, ou os exemplos um pouco artificiais, mas o objetivo foi sempre permitir ao utilizador aplicar o tipo de grelha que mais se adequar ao seu caso. Isso pode ser uma simples grelha onde só define o tamanho de cada célula, e os *ranges* são cada metade de cada célula, empurrando todos os eventos para o separador que lhe esteja mais próximo, ou exemplos de grelhas assimétricas, onde possivelmente os eventos são sempre movidos só numa direção (esquerda ou direita) ou onde eventos que estejam perto dos separadores não sejam movidos.

5.3.4 Teclados Musicais

Para além de permitir construir expressões que geram sequências de eventos musicais 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 musicais, referi-mo-nos à possibilidade de descrever na nossa linguagem mapeamentos entre *eventos* e expressões musicais ou ações a executar quando esses eventos acontecem.

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

Listagem 5.22: 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`.

5.3.4.1 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`, respetivamente).

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

Listagem 5.23: 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 incluídas como parâmetros deste evento.

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

MouseMove Evento ocorre sempre que o rato é movido. Pode ser útil para 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.

```
1 @keyboard {
2    [keyboard\MouseMove] ($x, $y): print( $x, $y );
3 };
```

Listagem 5.24: 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 retornar 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.

5.3.4.2 Modificadores de Eventos

Cada evento de um teclado tem associada uma ação: essa ação pode ter *size effects*, e opcionalmente retornar um evento musical (ou uma sequência de eventos). Sempre que o evento é despoletado, a ação é avaliada e caso retorne um objeto musical, 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 musicais) 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 musical 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 musical. 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 liberta-la para a nota, podemos usar os modificadores `hold extend` em conjunto.

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

Listagem 5.25: Aplicar o modificador `hold extend` a um teclado inteiro

5.3.4.3 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);
```

```
7     keyboard\pop_flags($__keyboard; 'hold'; 'extend');
8     $__keyboard
9  }
```

Listagem 5.26: 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.

```
1  $i = 0;
2
3  @keyboard {
4     for ($k in @[ 'a', 's', 'd', 'f' ]) {
5        if ($i > 0) {
6           { $ki = $i };
7           [$k] hold extend: print( $i, $ki );
8        };
9        { $i += 1 };
10    }
11 };
```

Listagem 5.27: 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.

5.3.4.4 Gravar e Reproduzir Performances

Uma vez criado um ou mais teclados, o som produzidos por eles é direcionado automaticamente para os *outputs* (por predefinição as colunas de som). No entanto, é possível instruir os teclados para guardarem num ficheiro de *performance* uma lista das teclas ativadas. Isto permite depois ao utilizador reproduzir uma performance gravada, ou até efetuar modificações a essa gravação.

```
1 keyboard\record( $file );
```

Listagem 5.28: Gravação de uma *performance*

Através da função descrita na listagem 5.28, é possível instruir os teclados para gravarem para um determinado ficheiro, cujo caminho é passado como argumento. Também é possível passar o valor none para a função, o que efetivamente termina a gravação da performance atual (se estiver a decorrer alguma). De notar que uma vez que os teclados suportam registar como ações, não só expressões musicais, mas também quaisquer blocos de códigos válidos, podemos chamar esta função na ação de uma das teclas do teclado. Desta forma é possível iniciar e parar a gravação durante a execução do programa.

```
1 keyboard\replay( $file, $delay = 0 );
```

Listagem 5.29: Reprodução de uma *performance*

Da mesma maneira que é possível gravar uma *performance*, é possível também reproduzir uma *performance* a qualquer momento através da função referenciada em 5.29. Neste caso devemos passar o caminho do ficheiro contendo a *performance* gravada, bem como um parâmetro opcional (em milissegundos) para atrasar o início da reprodução durante algum tempo. Isto pode ser útil se quisermos tocar alguma faixa ao vivo por cima da gravação, para alinhar a mesma ou para servir de tempo de preparação.

```
1 $music = keyboard\readperf( $file, $keyboards = [] );
```

Listagem 5.30: Conversão de uma *performance* numa sequência de eventos músicais

Mas podemos não querer reproduzir diretamente uma *performance*. Também é possível carregar uma *performance* gravada para memória, como uma sequência de eventos musicais convencional, que podemos depois manipular com todas as funções e operadores que já vimos e que operam sobre música.

Esta função aceita também, para além do usual caminho do ficheiro de *performance*, uma lista opcional de teclados a usar na conversão. Quando a lista vai vazia, são utilizados todos os teclados ativos. Isto demonstra uma das funcionalidades mais interessantes da gravação de ficheiros de *performance*, na medida que permitem gravar os eventos das teclas ativadas, mas podemos depois reproduzir essas *performances* com teclados com notas e expressões diferentes associadas às teclas.

5.3.4.5 Buffers

Ao contrário de *performances*, que gravam os eventos que acionam as teclas dos teclados, é possível guardar diretamente os eventos musicais produzidos pelos teclados com a *class* `keyboard\Buffer`. Os eventos capturados em *buffers* podem ser tratados como qualquer outra sequência musical, 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 );
```

Listagem 5.31: 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 musical. 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 musical. 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 inicio.

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

from_music Permite substituir manualmente o conteúdo do *buffer* com uma sequência musical. 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 musicais, 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.

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

Listagem 5.32: 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.

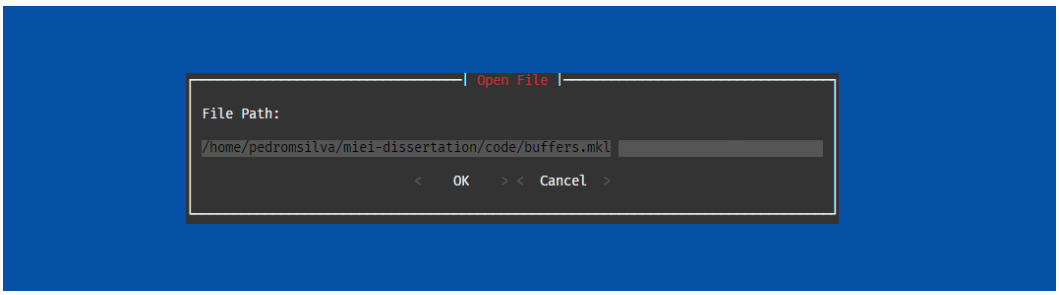


Figura 5.15: Janela de carregamento dos *buffers*

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))
```

Listagem 5.33: 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 à função `keyboard\Buffer.from_music()`.

5.3.5 Editor Embutido

O modo principal de execução de código baseia-se em carregar um ficheiro *Musikla* e executá-lo. Mas ao mesmo tempo, uma das funcionalidades mais importantes da linguagem é permitir descrever teclados musicais que permitem interação do utilizador enquanto a aplicação não for fechada.

No entanto, muitas das funcionalidades disponibilizadas pela aplicação são acessíveis através de classes e funções que podem ser chamadas pela linguagem. Isto tipicamente envolve uma de duas opções: o utilizador ter de mudar o ficheiro de código e executá-lo novamente, o que implica perder os dados que estejam em memória (como *buffers* e o possível estado do teclado). Também é possível associar pedaços de código a ações do teclado para permitir chamar esses pedaços de código durante a execução do programa, mas isso implica ser necessário planear todos os possíveis casos de uso à partida antes de iniciar executar o teclado. Se nos esquecermos de alguma, temos de voltar à opção inicial de mudar o código fonte e reiniciar a aplicação.

A solução passa pela inclusão de um simples editor de texto em *runtime* (implementado com recurso ao módulo *Python* `prompt_toolkit`. A biblioteca dispõem de uma função para permitir abrir e fechar o editor de texto ao premir uma tecla, ou também uma função para abrir diretamente o editor quando é chamada, dando assim ao utilizador mais controlo sobre como é possível usar este editor.

```
keyboard\repl( $key = "\\", $ctx = none );
```

Listagem 5.34: Abre o editor quando a tecla \ é premida

Na listagem 5.34 é possível ver a função que permite abrir o editor embutido. É também possível passar um contexto através do qual as expressões do editor irão ser avaliadas. Isto pode afetar quais as variáveis que são visíveis, ou até ter vários editores para diferentes contextos.

Figura 5.16: Interface do editor embutido.

```
keyboard\open_repl( $cb = none );
```

Listagem 5.35: Abrir manualmente o editor

Se não quisermos que o editor seja aberto simplesmente quando uma tecla é carregada, podemos chamar manualmente a função para o abrir. Isto tem a vantagem de podermos passar um *callback* ao editor para sermos notificados quando o editor é fechado. Isto significa que o editor pode ser usado não só como um ambiente de programação *Musikla* genérico, mas também como um modo de *input* de texto genérico a ser usado pelo utilizador.

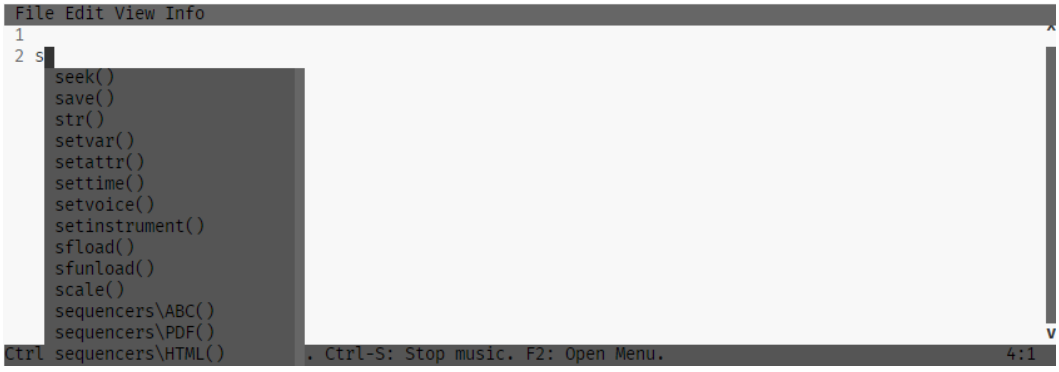


Figura 5.17: Implementação básica de uma solução de *auto-complete*.

Como prova de conceito, também foi incorporado um sistema de *auto-complete* que sugere símbolos que estejam declarados no nível base do contexto. Devido ao facto de o editor estar a correr ao mesmo tempo que a aplicação, esta funcionalidade não foi implementada através da análise dos ficheiros de código. Em vez disso, o objeto de contexto é analisado em tempo real para listar quais os símbolos que estão acessíveis.

Esta funcionalidade é extremamente útil tendo em conta que a linguagem tem como público alvo músicos que podem não ser programadores experientes. Por isso mesmo, ter um ambiente que assista o utilizador a saber quais as variáveis e métodos que pode utilizar.

No futuro, era interessante melhorar o *auto-complete* para completar sub-propriedades de objetos e listas, bem como mostrar informações adicionais sobre as sugestões, tal como os seus tipos e no caso de serem métodos, quais os parâmetros que aceita.

5.3.6 Transformadores

Os transformadores são uma solução para um problema prático que surgiu durante o desenvolvimento do projeto, ao invés de serem uma funcionalidade pública da linguagem. No entanto a razão pela qual foram necessários, bem como a solução desenvolvida, podem ser interessantes de serem discutidas. Para compreendermos melhor a razão da sua necessidade, imaginemos o seguinte caso.

5.3.6.1 O Problema

Um dos tipos de dados mais importantes na nossa linguagem é o tipo música, que como já referimos várias vezes pode ser representado como uma sequência de eventos musicais. Quando pensamos em sequências, é fácil imaginar uma variedade de operações genéricas possíveis de serem aplicadas nelas, como `map` e `filter`.

Mas também existem operações mais específicas ao nosso caso que podemos querer aplicar, como dada uma sequência de eventos, separar os `NoteEvent` em dois `NoteOnEvent` e `NoteOffEvent`. Isto à primeira vista pode parecer um simples `flatMap`, em que alguns eventos são transformados em dois a serem inseridos na sequência. No entanto, uma vez que os eventos nas nossas sequências devem viajar sempre ordenados, não podemos simplesmente inserir o evento *off* junto do *on*, porque podem existir outros eventos que comecem ao mesmo tempo ou depois da nossa nota começar, mas antes dela acabar.

Nesse caso basta construirmos uma algoritmo simples que guarde num *buffer* ordenado os eventos *off*, e os vá inserido imediatamente atrás do primeiro evento com um *timestamp* maior que os seus.

E como este, existem vários algoritmos que podem ser usados e reutilizados em diversas partes da aplicação com os mais diversos propósitos de transformar sequências de eventos.

O problema surge quando a fonte da música pode ser sincrona (no caso de uma expressão da nossa linguagem) ou assíncrona (no caso de um teclado ou piano, em que os eventos vão sendo gerados em tempo real). *Python* suporta assincronia através do módulo `asyncio` que pode ser usado em conjunto com a sintaxe *async/await*.

Para simplificar os exemplos seguintes, tomemos o caso da operação genérica *map*. As suas implementações, nas variantes síncrona e assíncrona, poderiam ser descritas como visto na listagem 5.37.

```
1 def map (fn):
2    for event in music:
3       yield fn(event)
4
5 async def map_async (fn):
6    async for event in music:
7       yield fn(event)
```

Listagem 5.36: Exemplo de uma função `map` com versões sincronas e asíncronas

Como podemos ver, quando a única diferença no algoritmo é se a **fonte é síncrona ou assíncrona**, a variação é mínima. Mas a diferença é sintática, por isso teríamos duas escolhas:

- Para cada algoritmo criar duas versões, uma síncrona e outra assíncrona. Mas isto levava a duplicação de código particularmente má, porque o que iria mudar eram apenas **duas palavras** em todas as funções.

- Usar sempre a versão assíncrona, mesmo quando a fonte dos dados é síncrona. Isto porque é sempre possível converter uma sequência de dados síncrona em assíncrona, mas o inverso já não é. O problema desta solução é o facto de as funções assíncronas terem *cor*[16], e só poderem ser chamadas por funções da mesma cor. Qualquer função que potencialmente lidasse com expressões musicais teria de passar a ser assíncrona. E como a nossa linguagem tem tipos dinâmicos, potencialmente qualquer expressão pode retornar uma sequência musical, logo **todas as expressões** na nossa linguagem teriam de ser executadas em modo assíncrono, o que traria uma penalização a nível de *performance* difícil de justificar.

5.3.6.2 A Solução Escolhida

A solução passou por criar uma abstração a que chamamos **Transformador**, que contém um bocado das duas opções anteriores: Os algoritmos são declarados apenas uma vez, e são depois envoltos num *transformador* que disponibiliza duas interfaces públicas, uma síncrona e outra assíncrona. Cada transformador tem uma interface pública composta por 4 funções que permitem simular um iterador: `add_input( ev )`, `end_input()`, `add_output( ev )` e `end_output()`.

Para evitar chamar manualmente as funções e tratar do controlo do fluxo de execução manualmente, são incluídos dois métodos de classe em todos os transformadores que permitem correr um iterador (síncrono ou assíncrono). As funções, chamadas `iter` e `aiter` recebem (e retornam) respetivamente um iterador síncrono e assíncrono.

Vejamos agora um exemplo do transformador **map** na listagem 5.37. Pode parecer bastante mais verbosa do que as funções *map* iniciais, mas é importante notar que este custo é fixo: só esta parte do algoritmo muda, e todo o resto seria igual. Como os transformadores utilizados no projeto são bem mais complexos do que este pequeno exemplo (podendo ter mais de uma centena de linhas de código), este custo fixo de escrever cerca de seis linhas de código *boilerplate* por transformador não é um problema tão grande.

```
1 class MapTransformer(Transformer):
2     def __init__ (self, fn):
3         self.fn = fn
```

[16] https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

```
4
5     def transform ( self ):
6         while True:
7             done, value = yield
8
9             if done: break
10
11            self.add_output( self.fn( value ) )
```

Listagem 5.37: Implementação da função `map` usando a nossa abordagem de transformadores

Todos os transformadores implementam uma função gerador *transform*, que é a peça central de tudo isto. Aí dentro, temos um ciclo responsável por emular a iteração sobre a sequência de *input*. Depois utilizamos a instrução *yield* para obter o próximo valor. Aqui estamos a aproveitar o facto de esta instrução não permitir apenas enviar valores para fora do gerador (que é o seu uso mais comum), mas também receber valores. Desta forma o gerador fica em pausa até receber o próximo evento. A vantagem desta pausa é que pode ser retomada logo de seguida se estivermos a trabalhar com iteradores síncronos, ou pode ficar assim até o próximo valor ficar disponível, no caso de um iterador assíncrono.

```
1 # Irá imprimir 0, 2, 4, 6, 8
2 for ev in MapTransformer.iter( range( 5 ), lambda e: e * 2 ):
3     print( ev )
```

Listagem 5.38: Exemplo de um transformador `map` e da sua utilização

Na listagem 5.38 podemos ver a utilização do transformador aplicada a uma lista (já que também implementam a interface de um iterador). O método `aiter` poderia ser usado de forma análoga com um *async for* e com um iterador assíncrono.

5.4 Resumo do Desenvolvimento

O processo de desenvolvimento foi bastante ágil, sendo estruturado de uma forma iterativa, em que a cada semana eram implementadas novas funcionalidades. Estas podiam depois ser postas à prova, aplicadas a casos de estudo, e possivelmente melhoradas numa iteração futura.

O tempo dedicado a cada um dos componentes da linguagem Musikla foi intercalado entre a análise sintática, semântica dinâmica e a biblioteca *standard*. Nem todas as funcionalidades exigiram esforço ou tempo equivalentes, e apesar de linhas de código não mapearem perfeitamente para tempo despendido, permitem ter alguma noção aproximada do mesmo.

Obviamente sendo estes números apenas uma contagem das linhas atuais, não são uma representação total de tudo o que foi programado, pois não têm em conta partes do código que foram reescritas ao longo do desenvolvimento, como foi o caso da gramática e do reconhecedor sintático, por exemplo.

Componente	Linhas de Código	Percentagem
Análise Sintática	1090	11%
Semântica Dinâmica	5268	53%
Biblioteca Standard	3527	36%
Total	9885	100%

Tabela 5.4: Linhas de código (sem linhas vazias ou de comentário) do projeto Musikla

Para além destes componentes, foi desenvolvido para ser usado na documentação um utilitário em *JavaScript*[17] com cerca de 500 linhas de código, que permite descrever e gerar os diagramas sobre grelhas (também usados nesta dissertação). A documentação está escrita no formato *Markdown* e conta também com um pouco mais de 700 linhas de texto. Este número não pode no entanto ser comparado diretamente com as linhas de código, já que texto em prosa e código são conceitos diferentes. Ainda assim ajudam a ter ideia do que foi feito, para além da escrita da desta dissertação, do artigo publicado e do desenvolvimento do interpretador Musikla.

[17] Código disponível em https://github.com/pedromsilvapt/miei-dissertation/blob/master/code/musikla/docs/assets/generator.js

Capítulo 6

Guia Rápido de Utilização

O resultado final do processo de desenho e desenvolvimento descrito acima foi um módulo *Python* publicado no repositório *PyPi* chamado `musikla`. Este módulo pode ser usado através da linha de comandos, ou os seus sub-módulos podem ser usados diretamente num projeto *Python*.

Nesta secção vamos fazer uma pequena *overview* dos diversos modos de como o módulo pode ser utilizado (versão da documentação mais completa disponível em na web).

A instalação é bastante simples, sendo suficiente correr o comando seguinte comando:

```
pip install musikla
```

Listagem 6.1: Processo de instalação do *package* musikla

É importante notar que uma dependência central da linguagem é a biblioteca nativa *FluidSynth*. Apesar de existir uma versão `2.x`, essa não é suportada atualmente, pelo que é necessário o utilizador certificar-se que o seu sistema tem a versão `1.x` instalada.

6.1 Módulos

O projeto **musikla** está dividido em bastantes sub-módulos, os mais importantes dos quais são listados aqui:

musikla.core.theory Contém classes e funções auxiliares relativas à teoria músical, utilizadas por diversos outros sub-módulos.

musikla.core.events.transformers Alberga os transformadores responsáveis por separar notas musicais em eventos On/Off, por exemplo, ou por tentar agrupar notas concorrentes em acordes, se for o caso disso, entre muitos mais.

musikla.core.events Contém as classes que representam os eventos musicais em memória, desde os mais óbvios, como o `NoteEvent`, `ChordEvent` ou `RestEvent`, mas também outros menos óbvios como o `ControlChangeEvent`, por exemplo.

musikla.core Disponibiliza muitas das classes mais fundamentais para o funcionamento da linguagem, tais como `Context`, `Music`, `Instrument`, `Voice`, `SymbolsScope`, entre outros.

musikla.parser.abstract_syntax_tree Contém as classes que representam os diversos nós da AST.

musikla.parser Disponibiliza as classes `Parser`, responsáveis por criar uma AST com base numa *string* ou num ficheiro contendo código Musikla, bem como a classe `CodePrinter` responsável pelo processo oposto, transformando de volta uma AST numa *string* usando a sintaxe da linguagem.

musikla.audio.sequencers Alberga os diversos sequenciadores (formatos de saída) suportados nativamente pela linguagem.

musikla.audio Contém as classes `Player` e `InteractivePlayer` que permitem reproduzir sequências musicais para os sequenciadores.

musikla.libraries Disponibilizam as várias classes que definem as funções e símbolos expostos para a linguagem Musikla pela biblioteca *standard*.

musikla O módulo principal, contém as duas classes responsáveis por inicializar todos os sistemas necessários para a linguagem funcionar, `CliApplication` e `Script`.

6.2 Configuração

Quando a aplicação é inicializada, é opcionalmente carregado um ficheiro `musikla.ini` se este se encontrar na pasta *Home* do computador do utilizador. Este ficheiro permite configurar diversos dos valores por predefinição utilizados pela linguagem. A maioria destas propriedades pode ser substituída e costumizada em cada execução quando a linguagem é executada pelo terminal.

```
[Musikla]
soundfont = /path/to/soundfont.sf2
output = -o alsa
midi_input = midi input port name
path = colon : separated list of paths to search for when importing files
prelude = colon : separated list of paths to include in the prelude (what is available to all
    ↪ modules)
autoload = colon : separated list of paths to execute before the main file does

[FluidSynth.Settings]
audio.periods = 2
```

```
audio.period-size = 64
synth.sample-rate = 44100
```

Listagem 6.2: Exemplo de um ficheiro de configuração da linguagem

O valor da propriedade **soundfont** indica o caminho do ficheiro `.sf2` a usar pela linguagem para sintetizar os sons.

A propriedade **output** permite indicar qual ou quais os sequênciadores (e os seus parâmetros) a serem por predefinição utilizados quando o utilizador não especifica nenhum em particular. A sua sintaxe é explicada em mais detalhe no capítulo seguinte.

A propriedade **midi_input** permite indicar o nome da porta *MIDI* a usar quando algum teclado está à escuta de eventos *MIDI*.

A propriedade **path** permite listar um conjunto de caminhos que serão procurados sempre que o utilizador importe algum ficheiro global nos seus *scripts*.

A propriedade **prelude** indica uma lista de *scripts* Musikla a serem executados e carregados, e cujos símbolos exportados devem ficar disponíveis para todos os outros módulos.

A propriedade **autoload** é similar à `prelude`, mas os símbolos apenas são disponibilizados para o ficheiro que o utilizador estiver a correr diretamente (o ficheiro *entry point*, podemos dizer).

Na secção **FluidSynth.Settings** podemos alterar manualmente as propriedades passadas ao sequênciador *FluidSynth*. A lista de propriedades e valores suportados está disponível aqui[1].

6.3 Linha de Comandos

A linha de comandos pode ser executada através do comando `musikla`. Podemos executar um ficheiro passando o seu caminho como argumento. Também podemos correr `musikla -help` para vermos uma listagem dos argumentos suportados.

```
musikla file.mkl --soundfont custom_soundfont.sf2
   -o alsa --gain 1
   -o music.wav
   -o musikla_port -f midi --port
```

Listagem 6.3: Exemplo de um ficheiro de configuração da linguagem

O comando pode ter uma lista de seuquênciadores (ou *outputs*) para utilizar. Estes devem vir sempre no fim do comando, depois de todos os argumentos e opções globais, e são identificados pela opção `-o` ou `-output`. O tipo do sequênciador é geralmente inferido pelo valor passado ao output, mas caso haja ambiguidade, é possível identificar manualmente o formato com a opção `-f` ou `-format`. Também são passadas a esse *output* todas as opções que estiverem depois do `-o` e até ao fim do comando, ou até ao próximo `-o` encontrado. As opções aceites variam com o formato do *output*.

[1] FluidSettings.xml http://www.fluidsynth.org/api/fluidsettings.xml

Capítulo 7

Conclusão

O desenvolvimento do projeto envolve vários aspetos, desde o desenho da sintaxe e da gramática correspondente, até à geração de sons a serem guardados em ficheiros ou reproduzidos imediatamente em dispositivos áudio. Também engloba aspetos comuns em todas as linguagens de programação, em conjunto com questões mais específicas sobre como gerir o conceito de tempo na linguagem, de *laziness* para gerar apenas as notas necessárias, entre muitos outros.

Para além disso também abrange quais as ferramentas e as metodologias que 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. Quais as funções e as suas interfaces que são mais úteis disponibilizar logo à partida a todos os utilizadores.

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

O projeto foi 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.

Durante o desenvolvimento, tornou-se claro que criar música através de programação é uma forma extremamente poderosa de se trabalhar em muitas situações, mas não é uma solução perfeita para todos os casos. Com isto em mente, sentimos que a nossa escolha de incluir diretamente na nossa linguagem a possibilidade de programar teclados permite cobrir os momentos em que o utilizador precisa de uma forma mais interativa e imediata de tocar e compor músicas. Para além disso, as funcionalidades extra de permitir gravar e reproduzir as *performances* em *buffers* enquanto os teclados estão ativos, bem como a inclusão de um simples editor de código que permite escrever e executar código em *runtime*

ofereceram uma experiência de composição e experimentação com ágil e rápido feedback loop.

A nível de processamento de linguagem, sentimos que a escolha de usar um interpretador *tree-walk* favoreceu muito a prototipagem de funcionalidades na linguagem. Mas fica sempre aberta a possibilidade de implementação da linguagem com um *backend* alternativo, quer através de uma maquina virtual de *bytecode*, ou possivelmente até *convertendo* o código Musikla para código *Python* e usando as funcionalidades de execução em *runtime* do mesmo.

A coleção de métodos e classes disponibilizados pela linguagem, e mais especificamente, pela biblioteca *standard* desenvolvida para acompanhar a linguagem, já permite o desenvolvimento de *scripts* bastante complexos e interessantes, como pode ser visto em muitos dos casos de estudo. Ainda assim, é indiscutível que existem ainda uma enorme quantidade de métodos e primitivas ligadas à teoria musical ou à geração procedural de música por adicionar, pois tal é uma tarefa sempre em evolução.

Ficamos também satisfeitos por reutilizar padrões, técnicas e bibliotecas onde existem já soluções bem estabelecidas, tanto a nível de formatos de *input/output*, como o caso do formato *MIDI* ou *abc*, que nos permitem apanhar boleia de todo o ecossistema de ferramentas construído já à sua volta. Também a utilização de módulos como o FluitSynth, que nos permitiu não só fazer programas que escrevessem ou retornassem notação musical como resultado, mas tocassem as notas em *realtime*, com suporte para vários instrumentos, sem o utilizador necessitar de configurar nada externo à aplicação, tornou-a a nosso ver muito mais interessante. Por fim, a utilização de notação existente (como a do *abc notation*) dentro da nossa própria linguagem reduz imenso a curva de aprendizagem necessária para alguém (quer já seja músico ou não) se tornar proficiente com ela.

Tudo isto resulta, na nossa opinião, num projeto que não reinventa a roda, mas sim adota os standards existentes onde estes fazem sentido, e pode assim focar-se em colmatar as áreas que realmente faltavam no ecossistema: uma linguagem declarativa e dinâmica para descrição de acompanhamentos musicais, e da possibilidade de criação de teclados interativos que combinam sem fricção com o resto da linguagem.

E assim esperamos que este projeto sirva como uma fundação. Que permita a mais pessoas no futuro fazer como nós fizemos, aproveitando as ideias que já existam e encontrem novas aberturas na área da música computacional. E desta vez aproveitando boleia da nossa linguagem, como nós fizemos também, criem novos projetos, novas ideias e novas possibilidades.

7.1 Trabalho Futuro

A linguagem e a aplicação Musikla foi um projeto bastante ambicioso, e com bastante potencial, não só pelo que já foi feito, mas pela enorme quantidade de funcionalidades que seriam extremamente interessantes de adicionar ainda.

Introduzir Percussão

Algo que ainda não foi bastante explorado mas que é bastante importante passa pela descrição de elementos de percussão na linguagem. Seria extremamente desejável estudar qual a sintaxe e a semântica mais ergonómica para esse efeito.

Processamento de Áudio Digital

Também era importante fornecer de raiz a possibilidade de o utilizador empregar técnicas de processamento de áudio digital para aplicar efeitos em tempo real aos sons gerados pela aplicação. Para isso, seria interessante estudar possíveis formas de integração com alguma das linguagens já mencionadas nesta dissertação (como a linguagem **FAUST**, por exemplo) para esse efeito. Esta abordagem teria a vantagem de utilizar ferramentas já otimizadas para a tarefa (que pode ser bastante exigente em termos computacionais), e permitir que utilizadores mais profissionais pudessem usar a nossa linguagem sem a terem de ligar a soluções externas (algo que é possível fazer, usando por exemplo a nossa saída MIDI para controlar algum Digital Audio Workstation (DAW) com a nossa linguagem).

Criação de Interfaces Gráficas

A linguagem e a aplicação Musikla atualmente fornecem só uma interface de linha de comandos, que apesar de bastante poderosa, pode ser limitativa em alguns casos. Gostávamos por isso de investigar quais as formas em como poderíamos permitir, dentro da nossa linguagem, ao utilizador descrever interfaces gráficas customizadas, e usá-las para receber *feedback* sobre o estado da aplicação ou controlar a música que é gerada.

Refactor do Interpretador

O interpretador da semântica dinâmica também poderia ser melhor otimizado caso o fator da *performance* se tornasse mais relevante (o que poderia acontecer no caso das interfaces gráficas). Para isso poderíamos converter em *runtime* o código para *Python* que seria depois mais rápido a executar, ou construir alguma máquina de *bytecode* que seria também mais rápida que a nossa solução atual.

Aumentar a Biblioteca *Standard*

A nível da biblioteca *standard* e dos formatos de entrada e saída que são suportados, trata-se de um esforço contínuo para adicionar novas funcionalidades, novos formatos, ou mesmo melhorar aqueles que são já suportados mas dando-lhes novas configurações e novas possibilidades de utilização.

Bibliografia

Aaron, S. (2016). Sonic pi – performance in education, technology and art. *International Journal of Performance Arts and Digital Media*, *12*(2), 171–178. doi:10.1080/14794713.2016.1227593. eprint: https://doi.org/10.1080/14794713.2016.1227593

ABC Notation. (s.d.). Official website for the ABC Notation project, includes a multitude of examples of valid .abc files and syntax. Obtido de http://abcnotation.com/

ABC Notation Examples. (2011). Obtido de http://abcnotation.com/examples

ABC Notation Standard v2.1. (2011). Formal description of the ABC Notation Standard. Obtido de http://abcnotation.com/wiki/abc:standard:v2.1

alda. (s.d.). Official webpage for the alda language, containing examples and documentation. Obtido de https://alda.io/

Dangling else. (s.d.). Description of the common ambiguity problem in programming languages' grammars. Obtido de https://en.wikipedia.org/wiki/Dangling_else

FAUST Examples. (s.d.). Very rich examples demonstrating the possibilities and ease-of-use of the language. Obtido de https://faust.grame.fr/doc/examples/index.html

FAUST Libraries. (s.d.). A comprehensive list of the libraries and their respective functions that are included with the FAUST language's runtime. Obtido de https://faust.grame.fr/doc/libraries/

FAUST Quick Start. (s.d.). Succinct introduction to the language FAUST and it's core concepts. Obtido de https://faust.grame.fr/doc/manual/index.html#quick-start

FAUST Targets. (s.d.). Description of the vast languages, formats and platforms FAUST can compile to. Obtido de https://faust.grame.fr/doc/manual/#a-quick-tour-of-the-faust-targets

FluidSynth. (s.d.). Obtido de http://www.fluidsynth.org/

FluidSynth Settings. (s.d.). Extensive list of all accepted FluidSynth settings, their possible values and descriptions. Obtido de http://www.fluidsynth.org/api/fluidsettings.xml

Frühauf, J., Kopiez, R. & Platz, F. (2013). Music on the timing grid: The influence of microtiming on the perceived groove quality of a simple drum pattern performance. *Musicae Scientiae*, *17*(2), 246–260. doi:10.1177/1029864913486793. eprint: https://doi.org/10.1177/1029864913486793

Gonzato, G. (2019). *Making music with abc 2*.

Helmholtz Pitch Notation. (s.d.). Explanation about the textual musical notation that is the basis of Musikla and ABC. Obtido de https://en.wikipedia.org/wiki/Helmholtz_pitch_notation

Henningsson, D. & Team, F. (2011). Fluidsynth real-time and thread safety challenges. Em *Proceedings of the 9th international linux audio conference, maynooth university, ireland* (pp. 123–128).

McCartney, J. (2002). Rethinking the computer music language: Supercollider. *Computer Music Journal*, *26*(4), 61–68. doi:10.1162/014892602320991383. eprint: https://doi.org/10.1162/014892602320991383

Orlarey, Y., Fober, D. & Letz, S. (2009). FAUST : an Efficient Functional Approach to DSP Programming. Em E. D. FRANCE (Ed.), *NEW COMPUTATIONAL PARADIGMS FOR COMPUTER MUSIC* (pp. 65–96). Obtido de https://hal.archives-ouvertes.fr/hal-02159014

Orlarey, Y., Graef, A. & Kersten, S. (2006). DSP Programming with Faust, Q and SuperCollider. Em LAC (Ed.), *Linux Audio Conference*. Karlsruhe, Germany. Obtido de https://hal.archives-ouvertes.fr/hal-02158894

Rossum, D. & Joint, E. (1995). The soundfont® 2.0 file format. *Joint E-Mu/Creative Tech Center white paper*.

Sonic Pi. (s.d.). Comprehensive list of articules, examples, talks and many more resources regarding the Sonic Pi language. Obtido de https://sonic-pi.net/

Sonic Pi Fx Cheatsheet. (s.d.). Page documentating the effects and their properties available to use with Sonic Pi. Obtido de https://github.com/samaaron/sonic-pi/blob/master/etc/doc/cheatsheets/fx.md

Soundfont technical specification. (2006). Obtido de http://www.synthfont.com/sfspec24.pdf

TiMidity++. (s.d.). Obtido de http://timidity.sourceforge.net/

Wang, G., Cook, P. R. & Salazar, S. (2015). Chuck: A strongly timed computer music language. *Computer Music Journal*, *39*(4), 10–29. doi:10.1162/COMJ_a_00324. eprint: https://doi.org/10.1162/COMJ_a_00324

Wang, G., Cook, P. R. et al. (2003). Chuck: A concurrent, on-the-fly, audio programming language. Em *Icmc*.

Whitehead, N. (2019). Pyfluidsynth. Bindings for the FluidSynth project for the Python language. GitHub. Obtido de https://github.com/nwhitehead/pyfluidsynth

WildMidi. (s.d.). Obtido de https://www.mindwerks.net/projects/wildmidi/

Apêndice A

Gramática PEG

```
main <- body EOF;

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

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

import <- "import" _ namespaced
      / "import" _ string_value
      ;

var_declaration <- expression _ var_declaration_infix? "=" _ expression;

var_declaration_infix <- "+" / "-" / "/" / "*" / "|" / "&";

voice_declaration <- !"::" ":" namespaced _ "=" _ voice_declaration_body;

voice_declaration_body <- !"::" ":" namespaced _ "(" _ expression _ ")"
                  / expression
                  ;

function_declaration <- "fun" _ namespaced? _ "(" _ arguments? _ ")" _ "{" body "}"
                / "fun" _ namespaced? _ "(" _ arguments? _ ")" _ "=>" _ expression
                ;

```

```
arguments <- single_argument ( _ arguments_separator _ single_argument )*;

single_argument <- single_argument_prefix _ "$" identifier _ "=" _ expression
             / single_argument_prefix _ "$" identifier
             ;

single_argument_prefix <- "expr" / "ref" / "in" / "";

arguments_separator <- ',' / ';';

for_variables <- variable ( _ arguments_separator _ variable )*;

for_loop_head <- "for" _ "(" _ for_variables _ "in" _ value_expression _ ".." _
    ↪ value_expression _ ")"
            / "for" _ "(" _ for_variables _ "in" _ value_expression _ ")"
            ;

for_loop_statement <- for_loop_head _ "{" _ body? _ "}";

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

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

return_statement <- "return" _ expression?;

// BEGIN Keyboard
keyboard_declaration <- "@keyboard" (_ alphanumeric)* ( _ group )? _ "{" _ keyboard_body _ "}"
    ↪ ;

keyboard_body <- keyboard_body_statement ( _ ";" _ keyboard_body_statement )* _ ";"?
          / ""
          ;

keyboard_body_statement <- keyboard_for
                  / keyboard_while
                  / keyboard_if
                  / keyboard_block
                  / keyboard_shortcut
                  ;

keyboard_for <- for_loop_head _ "{" _ keyboard_body _ "}";

keyboard_while <- "while" _ "(" _ expression _ ")" _ "{" _ keyboard_body _ "}";
```

```
71
72  keyboard_if <- "if" _ "(" _ expression _ ")" _ "{" _ keyboard_body _ "}" _ "else" _ "{" _
        ↪ keyboard_body _ "}"
73            / "if" _ "(" _ expression _ ")" _ "{" _ keyboard_body _ "}"
74            ;
75
76  keyboard_block <- "{" _ body _ "}";
77
78  keyboard_shortcut <- keyboard_shortcut_key _ keyboard_arguments? _ ":" _ expression
79                ;
80
81  list_comprehension <- expression "for" _ "$" _ namespaced _ "in" _ value_expression _ ".." _
        ↪ value_expression
82                / expression "for" _ "$" _ namespaced _ "in" _ value_expression _ ".." _
                      ↪ value_expression _ "if" value_expression
83                ;
84
85  keyboard_shortcut_key <- alphanumeric (_"+"_ alphanumeric)* (_ alphanumeric)*
86                    / string_value (_ alphanumeric)*
87                    / "[" _ list_comprehension _ "]" (alphanumeric _)*
88                    / "[" _ value_expression _ "]" (alphanumeric _)*
89                    ;
90
91  keyboard_arguments <- "(" _ ( keyboard_single_argument ( _ arguments_separator _
        ↪ keyboard_single_argument )* )? _ ")";
92
93  keyboard_arguments_separator <- ',' / ';';
94
95  keyboard_single_argument <- "$" identifier;
96  // END Keyboard
97
98  python_expression <- "@py" _ "{" _ python_expression_body _ "}";
99
100 python_expression_body <- r"[^\}]*";
101
102 python_statement <- "@python" python_statement_body;
103
104 python_statement_body <- ( r".*" r"\r?\n"? )*;
105
106 expression <- e music_expression;
107
108 music_expression <- sequence ( _ "|" _ sequence )*;
109
110 sequence <- value_expression ( _ value_expression)*;
111
```

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

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

variable <- "$" namespaced;

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

function_parameters <- named_parameters
                / positional_parameters ( _ parameters_separator _ named_parameters )?
                / e
                ;

positional_parameters <- !named_parameter expression ( _ parameters_separator _ !
    ↪ named_parameter expression )*;

named_parameters <- named_parameter ( _ parameters_separator _ named_parameter )*;

named_parameter <- identifier _ '=' _ expression;

parameters_separator <- ',' / ';';

note <- note_pitch note_value?;

chord <- "[" _ note_pitch chord_suffix _ "]" note_value?
     / "[" ( _ note_pitch )+ _ "]" note_value?
     ;

chord_suffix <- 'm7' / 'M7' / 'dom7' / '7' / 'm7b5' / 'dim7' / 'mM7'
           / '5'
           / 'M' / 'm' / 'aug' / 'dim' / '+'
           ;

rest <- "r" note_value?;

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

note_pitch <- note_accidental _ note_pitch_raw;

note_accidental <- "^" / "^^" / "__" / "_" / "";

```

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

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

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

value_expression <- e binary_logic_operator_expression;

binary_logic_operator_expression <- binary_comparison_operator_expression _ ("and" / "or") _
    ↪ binary_logic_operator_expression
                        / binary_comparison_operator_expression
                        ;

binary_comparison_operator_expression <- binary_sum_operator_expression _ (">=" / ">" / "==" /
    ↪  "!=" / "<=" / "<" / "isnot" / "is" / "in" / "notin") _
    ↪ binary_comparison_operator_expression
                            / binary_sum_operator_expression
                            ;

binary_sum_operator_expression <- binary_mult_operator_expression _ r"[+\-]" _
    ↪ binary_sum_operator_expression
                       / binary_mult_operator_expression
                       ;

binary_mult_operator_expression <- unary_operator_expression _ r"(\*\*|\*|\/)" _
    ↪ binary_mult_operator_expression
                        / unary_operator_expression
                        ;

unary_operator_expression <- ( "not" / "" ) _ expression_single;

expression_single <- expression_single_prefix ( ( _ property_accessor ) / property_call )*;

```

```
194 expression_single_prefix <- function_declaration / string_value / number_value / bool_value /
        ↪ none_value / variable / function / keyboard_declaration / python_expression /
        ↪ array_value / object_value / group / block / chord / note / rest / modifier /
        ↪ instrument_modifier;
195
196 property_accessor <- "::" _ ( identifier / ( "[" _ expression _ "]" ) );
197
198 property_call <- "(" _ function_parameters _ ")";
199
200 array_value <- "@[" _ "]"
201           / "@[" _ expression ( _ array_separator _ expression )* _ "]"
202           ;
203
204 array_separator <- ',' / ';';
205
206 object_value <- "@{" _ "}"
207            / "@{" _ object_value_item ( _ object_separator _ object_value_item )* _ "}"
208            ;
209
210 object_value_item <- object_value_key _ '=' _ expression;
211
212 object_value_key <- identifier / float / integer / double_string / single_string;
213
214 object_separator <- ',' / ';';
215
216 string_value <- double_string / single_string;
217
218 double_string <- "\"" double_string_char* "\"";
219
220 double_string_char
221    <- "\\\""
222     / "\\\\"
223     / r"[^\"]"
224     ;
225
226 single_string <- "'" single_string_char* "'";
227
228 single_string_char
229    <- "\\'"
230     / "\\\\"
231     / r"[^']"
232     ;
233
234 number_value <- float / integer;
235
```

```
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]*";

__ <- r"[ \t\r\n]+";

e <- "";

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

Listagem A.1: Gramática PEG da linguagem Musikla

Apêndice B

Gramática LALR

```
start: [body] [python]

?body: statement (";" statement)* ";"?

python: "@python" PYTHON_STATEMENTS

?statement: voice_assignment
        | import
        | for_loop_statement
        | while_loop_statement
        | return_statement
        | assignment
        | expression

assignment: expression ASSIGNMENT_OP expression -> assignment
        | expression ("," expression)+ ","? ASSIGNMENT_OP expression -> multi_assignment

voice_assignment: VOICE_IDENTIFIER "=" voice_assignment_body

voice_assignment_body: VOICE_CALL expression _RPAR -> voice_assignment_inherit
                | expression -> voice_assignment_base

import: "import" IDENTIFIER -> import_global
     | "import" STRING -> import_local

for_variables: VARIABLE_NAME ( _arguments_sep VARIABLE_NAME )*

for_loop_head: _FOR _LPAR for_variables "in" expression_atom ".." expression_atom _RPAR ->
    ↪ for_loop_head_range
```

```
          | _FOR for_variables "in" expression_atom ".." expression_atom ->
              ↪ for_loop_head_range
          | _FOR _LPAR for_variables "in" expression _RPAR -> for_loop_head
          | _FOR for_variables "in" logic_op_expr -> for_loop_head

for_loop_statement: for_loop_head if_body

while_loop_statement: _WHILE logic_op_expr if_body

if_statement: _IF logic_op_expr if_body _ELSE logic_op_expr -> if_statement_else
      // ¦ _IF logic_op_expr if_body _ELSE if_statement -> if_statement_else
         | _IF logic_op_expr if_body -> if_statement

if_body: "then"? logic_op_expr
// ¦ block

return_statement: _RETURN expression?

?expression: sequence_expr
        | sequence_expr ("|" sequence_expr)+ -> parallel

?sequence_expr: logic_op_expr
           | logic_op_expr logic_op_expr+ -> sequence

?logic_op_expr: comparison_op_expr
          | logic_op_expr "and" comparison_op_expr -> and_logic_op
          | logic_op_expr "or" comparison_op_expr -> or_logic_op

?comparison_op_expr: sum_op_expr
              | comparison_op_expr ">=" sum_op_expr -> gte_comparison_op
              | comparison_op_expr ">" sum_op_expr -> gt_comparison_op
              | comparison_op_expr _EQ sum_op_expr -> eq_comparison_op
              | comparison_op_expr "!=" sum_op_expr -> neq_comparison_op
              | comparison_op_expr "<=" sum_op_expr -> lte_comparison_op
              | comparison_op_expr "<" sum_op_expr -> lt_comparison_op
              | comparison_op_expr _ISNOT sum_op_expr -> isnot_comparison_op
              | comparison_op_expr _IS sum_op_expr -> is_comparison_op
              | comparison_op_expr _IN sum_op_expr -> in_comparison_op
              | comparison_op_expr _NOTIN sum_op_expr -> notin_comparison_op

?sum_op_expr: mult_op_expr
         | sum_op_expr "+" mult_op_expr -> sum_op
         | sum_op_expr "-" mult_op_expr -> sub_op

?mult_op_expr: unary_op_expr
```

```
          | mult_op_expr "**" unary_op_expr -> pow_op
          | mult_op_expr "*" unary_op_expr -> mult_op
          | mult_op_expr "/" unary_op_expr -> div_op

?unary_op_expr: expression_atom
           | "not" expression_atom -> negation

?expression_atom: expression_atom "::" accessor -> accessor
             | if_statement
             | function_declaration
             | string_literal
             | number_literal
             | bool_literal
             | none_literal
             | variable
             | function
             | keyboard_declaration
             | python_expression
             | array_literal
             | object_literal
             | group
             | block
             | chord
             | note
             | rest
             | modifier
             | voice_modifier

accessor: FUNCTION_CALL function_call -> method_call
      | IDENTIFIER -> property_accessor
      | _LSQR expression _RSQR -> index_accessor
      | _LSQR expression _INDEX_CALL function_call -> index_call

python_expression: "@py" _LBRAK PYTHON_EXPR _RBRAK

// Function Calls
function: FUNCTION_CALL function_call

function_call: [function_parameters] _RPAR -> function_call
          | [function_parameters] _GROUP_CALL function_call -> function_call_chain

function_parameters: parameter (_parameters_sep parameter)*

?parameter: named_parameter | positional_parameter

```

```
named_parameter: NAMED_IDENTIFIER expression

positional_parameter: expression

_parameters_sep: "," | ";"
// END Function Calls


// BEGIN Keyboard
keyboard_declaration: "@keyboard" keyboard_flags (_LPAR expression _RPAR)? _LBRAK [
    ↪ keyboard_body] _RBRAK

keyboard_flags: ALPHANUMERIC*

keyboard_body: keyboard_body_statement ( ";" keyboard_body_statement )* ";"?

?keyboard_body_statement: keyboard_for
                  | keyboard_while
                  | keyboard_if
                  | keyboard_block
                  | keyboard_shortcut

keyboard_for: for_loop_head _LBRAK [keyboard_body] _RBRAK

keyboard_while: _WHILE _LPAR expression _RPAR _LBRAK [keyboard_body] _RBRAK

keyboard_if: _IF _LPAR expression _RPAR _LBRAK [keyboard_body] _RBRAK "else" _LBRAK [
    ↪ keyboard_body] _RBRAK
        | _IF _LPAR expression _RPAR _LBRAK [keyboard_body] _RBRAK "else" keyboard_if
        | _IF _LPAR expression _RPAR _LBRAK [keyboard_body] _RBRAK

keyboard_block: _LBRAK body? _RBRAK

keyboard_shortcut: keyboard_shortcut_key [keyboard_arguments] ":" expression

keyboard_shortcut_key: ALPHANUMERIC ("+" ALPHANUMERIC)* ALPHANUMERIC* ->
    ↪ keyboard_shortcut_key_static
                | STRING ALPHANUMERIC* -> keyboard_shortcut_key_string
                | _LSQR expression _RSQR ALPHANUMERIC* -> keyboard_shortcut_key_dynamic

keyboard_arguments: _LPAR ( VARIABLE_NAME ( _arguments_sep VARIABLE_NAME )* )? _RPAR

keyboard_arguments_separator: "," | ";"
// END Keyboard

```

```
// Function Declaration
function_declaration: _FUN [IDENTIFIER] _LPAR [arguments] _RPAR [using] _LBRAK [body] _RBRAK
    ↪ -> function_statements
                | _FUN [IDENTIFIER] _LPAR [arguments] _RPAR [using] _ARROW expression ->
                    ↪ function_expression

using: _USING VARIABLE_NAME (”,” VARIABLE_NAME)*

arguments: argument ( _arguments_sep argument )*

argument: [argument_prefix] VARIABLE_NAME ”=” expression -> argument_default
      | [argument_prefix] VARIABLE_NAME -> argument

!argument_prefix: ”expr” | ”ref” | ”in”

_arguments_sep: ”,” | ”;”
// END Function Declaration

block: _LBRAK body? _RBRAK
    | _LBRAK body? _BLOCK_CALL function_call -> block_call

group: _LPAR expression _RPAR
    | _LPAR expression _GROUP_CALL function_call -> group_call

// Literals
string_literal: STRING
number_literal: INTEGER -> integer_literal
           | FLOAT -> float_literal

!bool_literal: _TRUE | _FALSE
none_literal: _NONE

array_literal: ”@[” _RSQR
         | ”@[” expression ( _array_sep expression )* _RSQR

_array_sep: ”,” | ”;”

object_literal: ”@{” _RBRAK
           | ”@{” object_item ( _object_sep object_item )* _RBRAK

object_item: object_key ”=” expression

object_key: IDENTIFIER
        | FLOAT
        | INTEGER
```

```
        | STRING

_object_sep: "," | ";"
// END Literals

// Music
chord: _LSQR chord_note_pitch CHORD_SUFFIX _RSQR note_value? -> chord_shortcut
    | _LSQR chord_note_pitch+ _RSQR note_value? -> chord_manual

chord_note_pitch: NOTE_ACCIDENTAL? NODE_PITCH_RAW -> note_pitch

CHORD_SUFFIX: "m3" | "M3"
          | "m7b5" | "m7" | "M7" | "dom7" | "7" | "dim7" | "mM7"
          | "5"
          | "M" | "m" | "aug" | "dim" | "+"

rest: "R"i note_value?

note: note_pitch note_value?

note_value: "/" INTEGER -> note_value_frac
        | INTEGER "/" INTEGER -> note_value_frac
        | INTEGER -> note_value_int

note_pitch: NOTE_ACCIDENTAL? NODE_PITCH_RAW

NOTE_ACCIDENTAL: "^^" | "^" | "__" | "_"

NODE_PITCH_RAW.-1: /[cdefgab]/ "'"*
           | /[CDEFGAB]/ ","*

modifier: "T"i INTEGER -> tempo_mod
       | "V"i INTEGER -> velocity_mod
       | "I"i INTEGER -> instrument_mod
       | "L"i note_value -> length_mod
       | "S"i INTEGER "/" INTEGER -> signature_mod
       | "S"i INTEGER -> signature_mod
       | "O"i INTEGER -> octave_mod

voice_modifier: VOICE_IDENTIFIER sequence_expr
VOICE_CALL: VOICE_IDENTIFIER "("
VOICE_IDENTIFIER: ":" IDENTIFIER
// END Music

variable: VARIABLE_CALL function_call -> variable_call
```

```
      | VARIABLE_NAME -> variable_name

FUNCTION_CALL.3: IDENTIFIER "("
VARIABLE_CALL.3: VARIABLE_NAME "("
_GROUP_CALL.2: ")("
_BLOCK_CALL.2: "}("
_INDEX_CALL.2: "]("

NAMED_IDENTIFIER.2: IDENTIFIER WS* "="
VARIABLE_NAME: "$" IDENTIFIER
IDENTIFIER.2: /[a-zA-Z_][a-zA-Z0-9_\\]*/
ALPHANUMERIC.2: /[a-zA-Z0-9_]+/

_FUN.4: "fun"
_USING.4: "using"
_WHILE.4: "while"
_FOR.4: "for"
_IF.4: "if"
_ELSE.4: "else"
_RETURN.4: "return"
_IN.4: "in"
_NOTIN.4: "notin"
_NOT.4: "not"
_AND.4: "and"
_OR.4: "or"
_ISNOT.4: "isnot"
_IS.4: "is"
_TRUE.4: "true"
_FALSE.4: "false"
_NONE.4: "none"

_LPAR: "("
_RPAR: ")"

_LBRAK: "{"
_RBRAK: "}"

_LSQR: "["
_RSQR: "]"

_EQ.2: "=="
_ARROW.3: "=>"

// Common
FLOAT.2: /[0-9]+\.[0-9]+/
```

```
INTEGER: /0|[\-\+]?[1-9]\d*/i
STRING: /("(?!"").*?(?<!\\)(\\\\)*?"|'(?!'').*?(?<!\\)(\\\\)*?')/is
COMMENT: /#[^\n]*/
PYTHON_STATEMENTS.-1: /.+/s
PYTHON_EXPR.-1: /[^}]+/
WS: /[ \t\r\n]/
ASSIGNMENT_OP: ("**" | "+" | "-" | "/" | "*" | "&" | "|")? "="


%import common.WORD // imports from terminal library
%ignore WS // Disregard spaces in text
%ignore COMMENT
```

Listagem B.1: Gramática LALR da linguagem Musikla