# 2021－2022 学年 第一学期期末试卷

# B 卷

学号　　姓名　　成绩

考试日期：　2021 年 12 月 27 日

## 考试科目：《程序设计语言原理》

## Musicode ——基于 C 的音乐编程语言

## 一、语言设计驱动

### 1 编程语言和音乐创作的类比

对于一种编程语言来说，通常的操作是将源代码通过词法分析、语法分析、中间代码生成、中间代码优化和生成目标代码五个基本步骤，生成可在目标机器上运行的目标代码。

而对于音乐创作来说，目标机器分为两类，一类目标机器是各式各样的乐器甚至是人的嗓音，引导这些“硬件”演奏出优美旋律的“机器码”是乐谱；另一类目标机器是音乐播放器，对应的机器码就是音乐播放器可以是别的二进制文件，如 mp3、wav 和 mid 等，如图 1 所示。

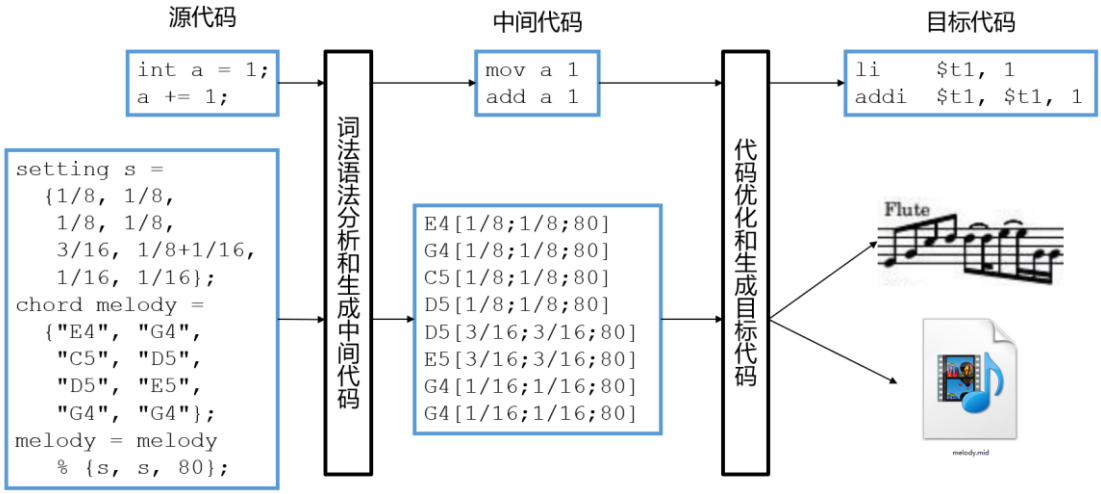


图 1 编程语言和音乐创作的对比图

Musicode 语言的源代码语法和 C 语言类似，可以帮助有一定编程基础的人快速方便地进行音乐创作。Musicode 编程语言将源代码，先通过编译器前端生成“音名&间隔”中间代码，然后通过编译器后端生成用于播放器播放音乐的音乐描述文件midi 以及 方便演奏者进行乐器演奏和方便演唱者进行演唱的五线谱。

## 2 利用乐理知识进行乐曲创作

与音乐翻译语言 alda 或是简单使用正则表达式进行音乐记录不同，Musicode 不仅仅可以更方便地记录乐谱，还可以帮助创作者利用乐理知识去创作音乐。当我们确定了一个特定的和弦进行（和弦就是多个音同时演奏，如最常见的 C 和弦就是将 do、mi 和 sol 同时演奏），如 C、C7sus4、G7、C，我们便得到了一系列柱式和弦，如图 2 所示。

```
chord C = "C";
chord C7sus4 = "C7sus4";
chord G7 = "G7";
G7 = G7 / 2;
chord res = C | C7sus4 | G7 - 12 | C;
play(res);
```

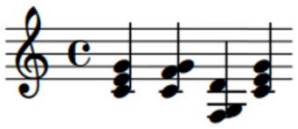

图 2 柱式和弦

此时仅仅确定了音乐的感情色彩、伴奏等基础信息，并没有确定音乐的主旋律，下面可以跟着和弦进行哼唱或用乐器演奏，找到一个与伴奏适配的旋律，这边完成了简单的乐曲创作。

```
setting s = {1, 2, 3, 1.1, 2.1, 3, 1.1, 2.1};

chord C = "C";
C = C @ s % {1/8, 1/8} * 2;

chord C7sus4 = "C7sus4";
C7sus4 = C7sus4 @ s % {1/8, 1/8} * 2;

chord G7 = "G7";
G7 = G7 / 2;
G7 = G7 @ s % {1/8, 1/8} * 2;

chord res = C | C7sus4 | G7 - 12 | C;
play(res);
```

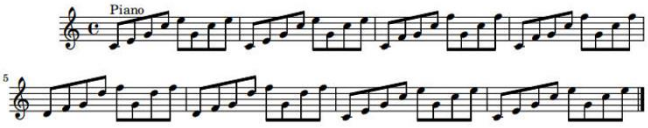


图 3 巴赫-前奏曲与赋格 C 大调第 1 号 BWV 846 分解和弦

图 3 的“巴赫-前奏曲与赋格 C 大调第 1 号 BWV 846”便是根据上述和弦进行分解演奏（取每个和弦的部分音）而创作出的乐曲。这个过程很难简单地使用翻译器实现。

### 3 作为 AI 作曲的语言接口

通常的 AI 作曲，不论是使用 CNN 或是 RNN 作为特征提取器，或是使用强化学习的方法，都是将乐曲片段或是 midi 文件作为输入。而 midi 这些文件单纯存储音符，力度，速度等单位化的信息，很难使网络学到乐理上的知识。我们希望创建一个从乐理上的角度来表示一段音乐从作曲上的角度是如何实现的语言。大部分的音乐都是极其具有乐理上的规律性的，这些规律抽象成乐理逻辑语句可以大大地精简化。

在人工智能高速发展的时代，我们希望作曲 AI 能在真正理解乐理的情况下进行乐曲创作，而不是使用深度学习，用大量的现成曲目进行训练，未来我们的 musicode 语言或许可以作为 AI 理解音乐的接口，使机器真正学到更深层次的知识。

我们希望设计一个类似于 C 语言文法的音乐编程语言 Musicode，使得熟悉 C 语言的人可以快速上手，使用它进行自己的音乐创作。Musicode 是一种静态语言，可以在编译时确定变量类型以及进行错误检查。Musicode 可以用非常简洁的语法来表达一段音乐的音符，和弦，旋律，节奏，力度等信息，可以通过乐理逻辑来生成曲子，并且进行高级的乐理操作。Musicode 除了用来创作音乐之外，还可以从乐理层面上来创作音乐和分析音乐，并且可以在 Musicode 的基础上设计乐理算法来探索音乐的可能性。

## 二、语言特性

### 1 文法

<乐段> → {(<表达式> | <播放语句> | <声明语句> | <打印语句>) ‘;’}

<标识符> → <字母> | <标识符><字母> | <标识符><数字>

<乐曲声明> → ‘piece’ <标识符> [ ‘=’ <配置>]

<配置列表> → <表达式> | ‘{’ <表达式> {‘,’ <表达式>} ‘}’

<配置> → <配置列表> | ‘{’ <配置列表> {‘,’ <配置列表>} ‘}’

<配置声明> → ‘setting’ <标识符> [ ‘=’ <配置>]

<音符声明> → ‘note’ <标识符> [ ‘=’ (<数> | <字符串>)]

<音符列表> → ‘{’ (<字符串> | <标识符>) {‘,’ (<字符串> | <标识符>)} ‘}’

<和弦声明> → ‘chord’ <标识符> [ ‘=’ (<字符串> | <音符列表>)]

<声明语句> → <乐曲声明> | <配置声明> | <音符声明> | <和弦声明>

<表达式> → <拼接表达式> | <一元表达式> <赋值运算符> <表达式>

<赋值运算符> → ‘=’ | ‘/=’ | ‘*=’ | ‘%=’ | ‘+=’ | ‘-=’ | ‘&=’ | ‘|=’

<拼接表达式> → <加法表达式>|<拼接表达式><拼接运算符><加法表达式>

<拼接运算符> → ‘|’ | ‘&’

<增减表达式> → <修改表达式>|<增减表达式><增减运算符><修改表达式>

<增减运算符> → ‘+’ | ‘-’

<修改表达式> → <一元表达式>|<修改表达式><修改运算符><一元表达式>

<修改运算符> → ‘/’ | ‘*’ | ‘**%**’ | ‘***@***’

<一元表达式> → <后缀表达式> | {‘+’ | ‘-’} [‘~’] <一元表达式> {‘+’ | ‘-’}

<后缀表达式> → <基本表达式> | <后缀表达式> ‘[’ <表达式> ‘]’

<基本表达式> → <标识符> | <配置> | <数> | <字符串> | ‘(’ <表达式> ‘)’

<播放语句> → ‘play’ ‘(’ <标识符> ‘)’ | ‘score’ ‘(’ <标识符> ‘)’

<打印语句> → ‘print’ ‘(’ <标识符> ‘)’

<数> → <数字> {<数字>} [ ‘.’ <数字> {<数字>}]

<字母> → A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<数字> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

## 2 新增数据类型

根据音乐描述的特点，我们创造了四种数据类型，用于更方便地抽象和表达音乐，如图 4 所示。


图 4 新增数据类型

其中 Setting 表示一个数，一个列表，或者列表的列表，用于%和@操作符对 note 或 chord 做出更改。Note 表示一个音符，其中包含了音名(C, E, Gb, ... 一个表示音名的字符串)，八度数(和音名一起确定一个音的音高)，时长(音符长度，单位为小节)和音量(音符的力度，范围为 0-127)等信息。Chord 被定义为一组音符的集合，这个定义或许比乐理里面的和弦定义更为广义化，因为按照这个定义，一首完整的乐曲也可以完全装进和弦类里面。Chord 包含了音符(音符列表，为一个记载着这个和弦所有音符的列表)，时长(和弦的每个音符各自的音符长度)和间隔(每两个连续音符之间的间隔，单位为小节)。 Piece 用于将多个 chord 合并起来，为每个 chord 设置不同的乐器，生成多音轨的音乐

# 三、指称语义

## 1 本语言使用到的域

**①基本域**

Number：域元素为实数

Character：域元素取自某字符集

Pitch-Name(音名) = C，C#(Db)，D，D#(Eb)，E，F，F#(Gb)，G，G#(Ab)，A，A#(Bb)，B

Register(音区) = 1，2，3，4，5，6，7，8

**②笛卡尔积域**

Pitch(音高) = Pitch-Name × Register

Note(音符) = Pitch × Number(时长)

**③联合域**

Tone(音)：Note(Pitch × Number) + Interval(Number)

Setting_List(配置列表) = Tone × … × Tone

Setting(配置) = Setting_List × … × Setting_List

**④序列域**

Chord(和弦)中的元素是选自域 Tone 中的元素的有限序列，即同构序列元素。或为 nil 元素，或为 x • s，其中元素 x ∈ Tone，序列 s ∈ Chord。

String(字符串)中的元素是选自域 Character 中的元素的有限序列。

**⑤存储域**

Store：所有存储状态的集合，元素为 sto

Location：存储单元，元素为 loci

Storable：可储值域，元素为 stble

**⑥环境域**

Environ：变量的作用域，本语言中只有全局作用域，元素为 env

Bindable：标识符束定的值，元素为 bdble

Identifier：标识符，元素为 I

## 3.2 语义域，语义函数及辅助函数：（S ∈ Statement，E ∈ Expression，D ∈ Declaration）

Value = note Note + chord Chord + string String + number Number + setting Setting

Storable = Value

Bindable = value Value + variable Location (值和变量是可束定体)

**①语句执行的语义函数**

execute：Statement → (Environ → Store → Store)

语句的执行是将 Statement 域中的元素(一条命令 S)映射为程序状态的改变，即某一环境 env 下的 sto 映射为另一 sto'。

execute〖S〗env sto = sto'

**②表达式求值的语义函数是**

evaluate：Expression → (Environ → Store → Value)

每个表达式的求值是将 Expression 域中的元素 E 映射为从环境中取出改变了状态值，即在某一环境 env 的 sto 下求值 value。

evaluate〖E〗 env sto= …

**③声明确立的语义函数**

elaborate: Declaration→ (Environ → Store → Environ × store)

每个声明的确立是将 Declaration 域中的元素 D 映射为将环境 env 下 sto 变为有了新束定的环境 env' × sto'。

elaborate〖D〗 env sto = (env', sto')

**④辅助函数**

Store = Location → ( stored Storable + undefined(未定义) + unused(未使用))

empty_store：Store (所有元素都映射为 unused 的 Store 元素)

*empty_store = λloc.unused*

allocate：Store → Store × Location (将 sto 映射为 sto'，其中某些 loc 从未使用变为未定义)

*allocate sto =*

***let*** *loc = any_unused_location (sto)* ***in***

*(sto [loc→ undefined]，loc)*

deallocate：Store × Location → Store (将 sto 映射为 sto'，某些 loc 变为未使用)

*deallocate (sto，loc) = sto [loc → unused]*

update：Store × Location × Storable → Store (将 sto 映射为 sto'，sto 中的 loc 单元更新为 stble 值)

*update (sto，loc，stble) = sto [loc →stored stble]*

fetch：Store × Location → Storable (从 sto 的 loc 单元上取出 stble 值)

```
fetch (sto，loc) =
    let stored_value (stored stble) = stble
        stored_value (undefined) = fail
        stored_value (unused) = fail
    in
        stored_value (sto(loc))
```

Environ = Identifier → (bound Bindable + unbound)

empty_environ：Environ (所有标识符均处于未束定状态的空环境)

*enpty_environ = λI. unbound*

bind：Identifier × Bindable → Environ (只要有标识符束定于可束定体，环境就改变了)

*bind(I,bdble) = λI'. if I'=I then bound bdble else unbound*

overlay：Environ × Environ → Environ (两束定环境组成新环境 evn + evn'，若某标识符在两环境中均有束定，则新环境中以后束定复盖先束定)

```
overlay (env'，env) =
    λI. if env'(I) /=unbound then env'(I) else env(I)
```

find：Environ × Identifier → Bindable (在环境 env 中找出标识符 I 所对应的可束定体 bdble)

```
find(env,I) =
    let bound_value(bound bdble) = bdble
        bound_value(unbound) = ⊥
    in
        bound_value(env(I))
```

coerce：Store × Bindable → Value (获取已束定变量的值)

*coerce(sto，find(env，I)) = fetch(sto，loc)*

## 3.3 各语法短语的语义

**①数和标识符的语义等式**

evaluate〖N〗env sto = double(valuation N)

evaluete〖I〗env sto = coerce(sto，find(env，I))

**②语句短语语义等式**

| |
|---|
| <语句> → <赋值语句><br>\| <声明语句> |

**执行赋值语句。**在 env sto 中对 E 求值，并放入临时变元 val 中; 再在 env 找出 I 束定的单元放于临时变元 loc 中; 最后以 sto，loc，val 作参数调用辅助函数 update，loc 中的值被修改，I 因而得赋值。

execute〖I = E〗env sto =

**let** val = evaluate E env sto **in**

**let** variable loc = find(env，I) **in**

update(sto，loc，val)

**执行声明语句。**在 env sto 中确立声明 D。由于确立 D 既改变了 env 又改变了 sto, 则将它们作为序偶取出，覆盖原有环境 env 和存储 sto。

execute〖D〗 env sto =

**let** (env', sto') = elaborate D env sto **in**

env', sto'

**连续执行多条语句。**连续执行两命令的语义是：第一命令 S1 在 env sto 下执行，不会改变束定的环境，只改变存储。S2 在 S1 改变了的存储(括号内执行结果)中执行。

execute〖S1; S2〗env sto =

execute S2 env(execute S1 env sto)

**③表达式语义等式**

| |
|---|
| <表达式> → <表达式> \| <表达式><br>\| <表达式> & <表达式> |

| <表达式> @ <表达式>

| <表达式> % <表达式>

| <表达式> * <表达式>

| <表达式> / <表达式>

| <表达式> + <表达式>

| <表达式> - <表达式>

| + <表达式>

| - <表达式>

**和弦的拼接“|”。**得到音乐片段 B 追加到音乐片段 A 之后的新的音乐片段 concatenate(A, B)。

evaluate〖E1 | E2〗env sto =
**let** chord c1 = evaluate E1 env sto **in**
**let** chord c2 = evaluate E2 env sto **in**
chord(concatenate ( c1，c2 ))

**和弦的合并“&”。**得到音乐片段 B 和音乐片段 A 合并之后的新的音乐片段 merge(A, B)。

evaluate〖E1 & E2〗env sto =
**let** chord c1 = evaluate E1 env sto **in**
**let** chord c2 = evaluate E2 env sto **in**
chord(merge ( c1，c2 ))

**和弦添加音符“+”。**得到音乐片段 A 添加音符 b 后的音乐片段 add(A, b)。

evaluate〖E1 + E2〗env sto =
**let** chord c1 = evaluate E1 env sto **in**
**let** note c2 = evaluate E2 env sto **in**
chord(add ( c1，c2 ))

**和弦去除音符“-”。**得到音乐片段 A 去除音符 b 后的音乐片段 minus(A, b)。

evaluate〖E1 - E2〗env sto =
**let** chord c1 = evaluate E1 env sto **in**
**let** note c2 = evaluate E2 env sto **in**
chord(minus ( c1，c2 ))

**和弦重复“*”。**得到音乐片段 A 重复 n 次后的音乐片段 repeat(A, n)。

evaluate〖E1 * E2〗env sto =
**let** chord c1 = evaluate E1 env sto **in**
**let** number c2 = evaluate E2 env sto **in**
chord(repeat ( c1，c2 ))

**和弦转位“/”。**得到音乐片段 A 中音符转 n 位的音乐片段 reverse(A, n)。

```
evaluate〖E1 / E2〗env sto =
        let chord c1 = evaluate E1 env sto in
        let number c2 = evaluate E2 env sto in
        chord(reverse ( c1，c2 ))
```

**和弦修改“%”。**得到音乐片段 A 根据配置 s 修改后的音乐片段 modify(A, s)。

```
evaluate〖E1 % E2〗env sto =
        let chord c1 = evaluate E1 env sto in
        let setting s2 = evaluate E2 env sto in
        chord(modify ( c1，s2 ))
```

**和弦配置“@”。**得到音乐片段 A 根据配置 s 配置后的音乐片段 setting(A, s)。

```
evaluate〖E1 @ E2〗env sto =
        let chord c1 = evaluate E1 env sto in
        let setting s2 = evaluate E2 env sto in
        chord(setting ( c1，s2 ))
```

**双目音符升调“+”。**得到音符 a 升 n 个半音后的音符 rise(a, n)。

```
evaluate〖E1 + E2〗env sto =
        let note c1 = evaluate E1 env sto in
        let number c2 = evaluate E2 env sto in
        note(rise ( c1，c2 ))
```

**单目音符升调“+”。**得到音符 a 升 1 个半音后的音符 rise(a)。

```
evaluate〖+ E1〗env sto =
        let note c1 = evaluate E1 env sto i
        note(rise ( c1 ))
```

**双目音符降调“-”。**得到音符 a 降 n 个半音后的音符 fall(a, n)。

```
evaluate〖E1 - E2〗env sto =
        let note c1 = evaluate E1 env sto in
        let number c2 = evaluate E2 env sto in
        note(fall ( c1，c2 ))
```

**单目音符降调“-”。**得到音符 a 降 1 个半音后的音符 fall(a)。

```
evaluate〖- E1〗env sto =
        let note c1 = evaluate E1 env sto i
        note(fall ( c1 ))
```

**双目和弦升调“+”。**得到音乐片段 A 中音符升 n 个半音后的音乐片段 rise(A,n)。

```
evaluate〖E1 + E2〗env sto =
        let chord c1 = evaluate E1 env sto in
```

**let** number c2 = evaluate E2 env sto **in**
chord(rise ( c1，c2 ))

**单目和弦升调“+”。**得到音乐片段 A 中音符升 1 个半音后的音乐片段 rise(A)。

evaluate〖+ E1〗env sto =
**let** chord c1 = evaluate E1 env sto **in**
chord(rise ( c1 ))

**双目和弦降调“-”。**得到音乐片段 A 中音符降 n 个半音后的音乐片段 fall(A,n)。

evaluate〖E1 - E2〗env sto =
**let** chord c1 = evaluate E1 env sto **in**
**let** number c2 = evaluate E2 env sto **in**
chord(fall ( c1，c2 ))

**单目和弦降调“-”。**得到音乐片段 A 中音符降 1 个半音后的音乐片段 fall (A)。

evaluate〖- E1〗env sto =
**let** chord c1 = evaluate E1 env sto **in**
chord(fall ( c1 ))

**将和弦看成数组时，数组变量的语义描述。**

component：Number × Chord → Note (和弦索引)

*component(number, nil) = ⊥*

*component(number, var · arrvar) =*

***if** number = 0 **then** var*

***else** component(prodecessor(number, arrvar))*

fetch_chord：Store × Identifier → Chord

*fetch_chord (sto，nil) = nil*

*fetch_chord (sto，var · arrvar) =*

*fetch(sto，var) · fetch_array (sto，arrvar)*

update_chord：Store × Identifier × Chord → Store

*update_chord (sto，nil，nil) = sto*

*update_chord (sto，var·arrvar，val·arrval) =*

***let** sto'= update (sto，var，val) **in***

*update_chord (sto'，arrvar，arrval)*

# 四、 创作过程

## 1 Musicode 源代码


选取北航校歌的一个乐句作为例子，该乐句有两个声部(melody1 和 melody2)，每个声部四个小节(s1, s2, s3 和 s4)，并添加一个伴奏(accompany)，得到如下的源代码：

```
//第一小节
setting s1 = {1/8, 1/8, 1/8, 1/8, 1/8+1/16, 3/16, 1/16, 1/16};
                                        //第一小节音符时长和间隔的配置
s1 = {s1, s1, 80};                       //{时长, 间隔, 音量}
chord bar1_melody1 = {"E4", "G4", "C5", "D5", "D5", "E5", "G4",
"G4"};                                   //第一小节一声部的音名
bar1_melody1 = bar1_melody1 % s1;       //对音名进行配置生成旋律
chord bar1_melody2 = {"E3", "G3", "G3", "G3", "G3", "C4", "E3",
"E3"};                                   //第一小节二声部的音名
bar1_melody2 = bar1_melody2 % s1;       //对音名进行配置生成旋律

//第二小节
setting s2 = {1/8, 1/8, 1/8, 1/16, 1/16, 1/8+1/16, 5/16};
                                       //第二小节音符时长和间隔的配置
s2 = {s2, s2, 80};                       //{时长, 间隔, 音量}
chord bar2_melody1 = {"A4", "A4", "A4", "C5", "C5", "D5", "D5"};
                                       //第二小节一声部的音名
bar2_melody1 = bar2_melody1 % s2;       //对音名进行配置生成旋律
chord bar2_melody2 = {"F3", "F3", "F3", "F3", "A3", "B3", "B3"};
                                        //第二小节二声部的音名
bar2_melody2 = bar2_melody2 % s2;       //对音名进行配置生成旋律
```

```
//第三小节
setting s3 = {1/4, 1/8, 1/16, 1/16, 1/8, 1/8+1/16, 1/16, 1/16,
1/16};
s3 = {s3, s3, 80};
chord bar3_melody1 = {"C5", "C5", "A4", "G4", "E4", "G4", "E4",
"E4", "G4"};
bar3_melody1 = bar3_melody1 % s3;
chord bar3_melody2 = {"A3", "A3", "E3", "E3", "C3", "E3", "C3",
"C3", "E3"};
bar3_melody2 = bar3_melody2 % s3;

//第四小节
setting s4 = {1/4, 1/8+1/16, 1/16, 1/8, 3/8};
s4 = {s4, s4, 80};
chord bar4_melody1 = {"A4", "G4", "E4", "G4", "D4"};
bar4_melody1 = bar4_melody1 % s4;
chord bar4_melody2 = {"F3", "E3", "C3", "D3", "B2"};
bar4_melody2 = bar4_melody2 % s4;

//将四小节旋律合并到一起
chord melody1 = bar1_melody1 | bar2_melody1 | bar3_melody1 |
bar4_melody1;
chord melody2 = bar1_melody2 | bar2_melody2 | bar3_melody2 |
bar4_melody2;

//添加伴奏
chord Cmaj = "Cmaj";                    //创建 C 和弦并进行配置
Cmaj = Cmaj @ {1, 2, 3, 1.1, 2.1, 3, 1.1, 2.1} % {1/8, 1/8};
chord Fmaj = "Fmaj";                    //创建 F 和弦并进行配置
Fmaj = Fmaj @ {1, 2, 3, 1.1} % {1/8, 1/8};
chord Gmaj1 = "Gmaj";                   //创建 G 和弦并进行配置
Gmaj1 = Gmaj1 @ {1, 2, 3, 1.1} % {1/8, 1/8};
chord Amin = "Amin";                    //创建小 A 和弦并进行配置
Amin = Amin @ {1, 2, 3, 1.1, 2.1, 3, 1.1, 2.1} % {1/8, 1/8};
chord Gmaj2 = "Gmaj";                   //创建 G 和弦并进行配置
Gmaj2 = Gmaj2 @ {1, 2, 3, 1.1, 2.1, 3, 1.1, 2.1} % {1/8, 1/8};
chord accompany = Cmaj - 12 | Fmaj - 24 | Gmaj1 - 24 | Amin - 24
| Gmaj2 - 12;                           //对和弦进行降调和拼接

piece c;                                //将两个和弦和伴奏并列生成乐曲
c = {{melody1, melody2, accompany}, {74, 69, 1}, 60};
play(c);                                //演奏乐曲并生成 midi 文件
score(c);                               //生成五线谱
```

## 2 中间代码

中间代码中每个单元的格式为 “音高[时长，间隔，音量]”，如下：

| |
|---|
| 音轨 1: E4[1/8;1/8;80], G4[1/8;1/8;80], C5[1/8;1/8;80], D5[1/8;1/8;80], D5[3/16;3/16;80], E5[3/16;3/16;80], G4[1/16;1/16;80], G4[1/16;1/16;80], A4[1/8;1/8;80], A4[1/8;1/8;80], A4[1/8;1/8;80], C5[1/16;1/16;80], C5[1/16;1/16;80], D5[3/16;3/16;80], D5[5/16;5/16;80], C5[1/4;1/4;80], C5[1/8;1/8;80], A4[1/16;1/16;80], G4[1/16;1/16;80], E4[1/8;1/8;80], G4[3/16;3/16;80], E4[1/16;1/16;80], E4[1/16;1/16;80], G4[1/16;1/16;80], A4[1/4;1/4;80], G4[3/16;3/16;80], E4[1/16;1/16;80], G4[1/8;1/8;80], D4[3/8;3/8;80] |
| 音轨 2: E3[1/8;1/8;80], G3[1/8;1/8;80], G3[1/8;1/8;80], G3[1/8;1/8;80], G3[3/16;3/16;80], C4[3/16;3/16;80], E3[1/16;1/16;80], E3[1/16;1/16;80], F3[1/8;1/8;80], F3[1/8;1/8;80], F3[1/8;1/8;80], F3[1/16;1/16;80], A3[1/16;1/16;80], B3[3/16;3/16;80], B3[5/16;5/16;80], A3[1/4;1/4;80], A3[1/8;1/8;80], E3[1/16;1/16;80], E3[1/16;1/16;80], C3[1/8;1/8;80], E3[3/16;3/16;80], C3[1/16;1/16;80], C3[1/16;1/16;80], E3[1/16;1/16;80], F3[1/4;1/4;80], E3[3/16;3/16;80], C3[1/16;1/16;80], D3[1/8;1/8;80], B2[3/8;3/8;80] |
| 音轨 3: C3[1/8;1/8;100], E3[1/8;1/8;100], G3[1/8;1/8;100], C4[1/8;1/8;100], E4[1/8;1/8;100], G3[1/8;1/8;100], C4[1/8;1/8;100], E4[1/8;1/8;100], F2[1/8;1/8;100], A2[1/8;1/8;100], C3[1/8;1/8;100], F3[1/8;1/8;100], G2[1/8;1/8;100], B2[1/8;1/8;100], D3[1/8;1/8;100], G3[1/8;1/8;100], A2[1/8;1/8;100], C3[1/8;1/8;100], E3[1/8;1/8;100], A3[1/8;1/8;100], C4[1/8;1/8;100], E3[1/8;1/8;100], A3[1/8;1/8;100], C4[1/8;1/8;100], G3[1/8;1/8;100], B3[1/8;1/8;100], D4[1/8;1/8;100], G4[1/8;1/8;100], B4[1/8;1/8;100], D4[1/8;1/8;100], G4[1/8;1/8;100], B4[1/8;1/8;100] |

## 3 目标代码

目标代码为相应的 midi 文件：temp.mid，以及五线谱。