**Musicode：基于C的音乐编程语言**

|  |  |  |
| --- | --- | --- |
| 姓名 | 学号 | 分工 |
| 何逸宸 | ZY2106342 | 文法设计，语义分析，前端代码，文档撰写 |
| 梁世俊 | ZY2106323 | 文法设计，前端代码，后端代码 |

目录

[1 语言设计驱动 3](#_Toc90223375)

[1.1 编程和音乐创作的类比 3](#_Toc90223376)

[1.2 Musicode的设计动机 3](#_Toc90223377)

[2 语言特性 3](#_Toc90223378)

[2.1 文法 3](#_Toc90223379)

[2.2 保留字及运算符 5](#_Toc90223380)

[2.2.1 保留字 5](#_Toc90223381)

[2.2.2 运算符 5](#_Toc90223382)

[2.3 中间代码 5](#_Toc90223383)

[2.4 目标代码 5](#_Toc90223384)

[2.4.1 音乐描述文件midi 5](#_Toc90223385)

[2.4.2 五线谱 5](#_Toc90223386)

[3 指称语义 6](#_Toc90223387)

[3.1 本语言使用到的域 6](#_Toc90223388)

[3.1.1 基本域 6](#_Toc90223389)

[3.1.2 笛卡尔积域 6](#_Toc90223390)

[3.1.3 联合域 6](#_Toc90223391)

[3.1.4 序列域 6](#_Toc90223392)

[3.1.5 存储域 6](#_Toc90223393)

[3.1.6 环境域 7](#_Toc90223394)

[3.2 语义域，语义函数及辅助函数：（S ∈ Statement，E ∈ Expression，D ∈ Declaration） 7](#_Toc90223395)

[3.2.1 语句执行的语义函数 7](#_Toc90223396)

[3.2.2 表达式求值的语义函数是 7](#_Toc90223397)

[3.2.3 声明确立的语义函数 8](#_Toc90223398)

[3.2.4 辅助函数 8](#_Toc90223399)

[3.3 各语法短语的语义 9](#_Toc90223400)

[3.3.1 数和标识符的语义等式 9](#_Toc90223401)

[3.3.2 语句短语语义等式 10](#_Toc90223402)

[3.3.3 表达式语义等式 11](#_Toc90223403)

[4 作曲过程 15](#_Toc90223404)

[4.1 Musicode源代码 15](#_Toc90223405)

[4.2 语法树 15](#_Toc90223406)

[4.3 中间代码 15](#_Toc90223407)

[4.4 目标代码 15](#_Toc90223408)

# 1 语言设计驱动

## 1.1 编程和音乐创作的类比

对于一种编译语言来说，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，各式各样的乐器甚至人的嗓音相当于硬件，而引导这些“硬件”演奏出优美旋律的，就是“机器码”——乐谱。通过Musicode编程语言，可以将类似C语言的源代码，先通过前端生成“音名&间隔”中间代码，然后通过后端生成用于描述音乐的midi文件。

## 1.2 Musicode的设计动机

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。

# 2 语言特性

## 2.1 文法

|  |  |  |
| --- | --- | --- |
| <乐段> | → | <语句块> |
| <标识符> | → | <字母> | <标识符><字母> | <标识符><数字> |
| <音符声明> | → | ‘note’ <标识符> ‘;’ |
| <音符更改> | → | <标识符> ‘%’ <音符参数> |
| <音符参数> | → | <**音名**> [‘,’ <数> ‘,’ <数>] |
| <和弦声明> | → | ‘chord’ <标识符> ‘;’ |
| <时长列表> | → | <数> | ‘[’ <数> { ‘,’ <数> } ‘]’ |
| <和弦更改> | → | <标识符> ‘%’ ‘(’ <时长列表> ‘,’ <时长列表> ‘)’ |
| <表达式> | → | <和弦表达式> { <拼接运算符><和弦表达式> }  | <音符> | <数> |
| <拼接运算符> | → | ‘**#**’ | ‘&’ |
| <和弦表达式> | → | <和弦> { <和弦运算符><和弦> } |
| <和弦运算符> | → | ‘**<**’ | ‘**>**’ | ‘\*’ | ‘|’ |
| <和弦> | → | { <运算符> } ( <标识符> | ~~<和弦切片>~~ ) [ <运算符><数> ]  | <和弦更改> | <标识符> { ‘+’ <标识符> } | ‘(’ <表达式> ‘)’ |
| <音符> | → | { <运算符> } ( <标识符> | <和弦索引> ) [ <运算符><数> ]  | <音符更改> |
| <运算符> | → | ‘+’ | ‘-’ |
| <和弦索引> | → | <标识符> ‘[’ <数> ‘]’ |
| <语句> | → | <赋值语句> | <声明语句> | ‘{’ <语句块> ‘}’  | ‘case’ <语句> ‘case’ <语句> { ‘case’ <语句> }  | ‘while’ <语句> | <播放语句> | <乐曲语句> | <打印语句> |
| <播放语句> | → | ‘play’ ‘(’ <标识符> ‘)’ ‘;’ |
| <乐曲语句> | → | <和弦列表><乐器列表><节拍><开始时间><轨道名称> |
| <和弦列表> | → | ‘[’ {<标识符>} ‘]’ |
| <乐器列表> | → | ‘[’ {<乐器>} ‘]’ |
| <节拍> | → | <数> |
| <开始时间> | → | ‘[’ {<数>} ‘]’ |
| <音轨列表> | → | ‘[’ {<轨道名称>} ‘]’ |
| <语句块> | → | { <语句> } |
| <赋值语句> | → | <标识符> ‘=’ <表达式> ‘;’ |
| <声明语句> |  | <音符声明> | <和弦声明> |
| <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 |

## 2.2 保留字及运算符

### 2.2.1 保留字

Musicode语言共有10个保留字，分别为：

|  |  |  |  |
| --- | --- | --- | --- |
| note | chord | play |  |
|  |  |  |  |
|  |  |  |  |

### 2.2.2 运算符

Musicode语言的运算符及优先级如下表(从高到低排列)：

|  |  |
| --- | --- |
| **符号** | **解释** |
| + - |  |
| < > | \* |  |
| # & |  |

## 2.3 中间代码

中间代码的格式为

## 2.4 目标代码

目标代码的格式为音乐描述文件midi。midi文件是一种用于记录音乐的二进制文件，与波形文件不同，midi文件不对音乐进行抽样，而是对音乐的每个音符记录为一个数字，所以与波形文件相比文件要小得多，可以满足长时间音乐的需要。MIDI标准规定了各种音调的混合及发音，通过输出装置可以将这些数字重新合成为音乐。

五线谱，是世界上通用的一种记谱法，通过在五根等距离的平行横线上标以不同时值的音符及其他记号来记载音乐。生成的midi文件可以通过MidiNotate等软件生成五线谱。

# 3 指称语义

## 3.1 本语言使用到的域

### 3.1.1 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

### 3.1.2 笛卡尔积域

Pitch(音高) = Pitch-Name × Register

Note(音符) = Pitch × Number(时长)

### 3.1.3 联合域

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

### 3.1.4 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。

### 3.1.5 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

### 3.1.6 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

Storable = Value

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

### 3.2.1 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’

### 3.2.2 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= …

### 3.2.3 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’)

### 3.2.4 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’，其中某些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

### 3.3.1 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))

### 3.3.2 语句短语语义等式

|  |  |  |
| --- | --- | --- |
| <语句> | → | <赋值语句>  | <声明语句>  | ‘{’ { <语句> } ‘}’  | ‘case’ <语句> ‘case’ <语句>  | ‘while’ <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)

**条件语句。**在env sto下执行条件语句，若全局执行次数(N)为1, 则在同一env sto下执行S1，若全局执行次数为2则执行S2。

execute〖**case** S1 **case** S2〗env sto =

**if** N = number 1

**then** execute S1 env sto

**else** execute S2 env sto

**循环语句。**执行while命令的语义是execute\_while函数的在evn sto之下执行结果。该函数是递归定义的。

execute〖while S〗=

**let** execute\_while env sto =

execute\_while env (execute S env sto)

**in**

execute\_while

### 3.3.3 表达式语义等式

|  |  |  |
| --- | --- | --- |
| <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>  | <表达式> - <表达式>  | + <表达式>  | - <表达式> |

**和弦的拼接“#”。**可以得到音乐片段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“&”。**可以得到音乐片段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“<”。**可以得到音乐片段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“>”。**可以得到音乐片段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“\*”。**可以得到音乐片段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 ))

**和弦添加休止符“|”。**可以得到音乐片段A后面添加n小节休止符后的音乐片段rest(A, n)。

evaluate〖E1 | E2〗env sto =

**let** chord c1 = evaluate E1 env sto **in**

**let** number c2 = evaluate E2 env sto **in**

chord(rest ( c1，c2 ))

**双目音符升调“+”。**可以得到音符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“+”。**可以得到音符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“-”。**可以得到音符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“-”。**可以得到音符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“+”。**可以得到音乐片段A中所有音符升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“+”。**可以得到音乐片段A中所有音符升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“-”。**可以得到音乐片段A中所有音符降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“-”。**可以得到音乐片段A中所有音符降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)*

# 4 作曲过程

## 4.1 Musicode源代码

## 4.2 语法树

## 4.3 中间代码

## 4.4 目标代码

(<class 'musicode.tree.expr\_nodes.Identifier'>,=,)

(<class 'musicode.tree.expr\_nodes.Identifier'>,%,)

(,(,(1,/,16),(1,/,16),(1,/,8),(7,/,8)))