Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

РОБОТА З МУЗИКОЮ В HASKELL

Текстова частина до курсової роботи
за спеціальністю «Комп'ютерні науки» 122

Керівник курсової роботи
кандидат фізико-математичних наук,
доцент Проценко В.С.

_____

(підпис)

«____» _____ 2021 р.

Виконав студент Грицюк О.О.

«____» _____ 2021 р.

Київ – 2021

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики,

кандидат фізико-математичних наук

Гороховський С. С.

_____

(підпис)

«____» _____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студента 3 курсу факультету інформатики

Грицюка Олександра Олександровича

Тема: Робота з музикою в Haskell

Зміст текстової частини до курсової роботи:

- Анотація
- Вступ
- Розділ 1. Опис інструментів
- Розділ 2. Алгоритмічна композиція
- Висновки
- Список використаних джерел

Дата видачі «____» __ _____ 2021 р.

Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Тема: Робота з музикою в Haskell

Календарний план виконання курсової роботи:

№	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1	Отримання завдання на курсову роботу	10.11.2020	
2	Пошук джерел за темою роботи	06.12.2020	
3	Ознайомлення з джерелами за темою роботи	10.01.2021	
4	Ознайомлення з бібліотекою Euterpea	31.01.2021	
5	Проектування програмної системи	28.02.2021	
6	Написання практичної частини курсової роботи	25.04.2021	
7	Написання текстової частини курсової роботи	09.05.2021	
8	Створення презентації	16.05.2021	
9	Захист курсової роботи	25.05.2021	

Зміст

Анотація

Дана робота присвячена алгоритмічній композиції (створенню музичних композицій за допомогою комп’ютерних алгоритмів) за допомогою мови програмування функціональної парадигми *Haskell*. У якості бібліотеки для полегшення представлення та обробки музичних фрагментів обрано *Euterpea*, яка надає необхідний інтерфейс для цього. Описано використання генераторів разом із монадами *State* та *IO* для роботи із псевдовипадковими величинами. Наведено класифікацію підходів штучного інтелекту для розв’язання даної задачі. Поєднання деяких з них, а саме: граматики, системи на основі знань та еволюційні алгоритми, використано для створення власної програмної системи для автономної генерації коротких музичних фрагментів без втручання людини.

Вступ

З розвитком технічного прогресу, людство все більше почало використовувати програмне забезпечення та автоматизацію для багатьох видів діяльності. Особливо варто зазначити різноманітні сфери творчості, адже митці отримали безмежні можливості для обробки своїх творів, а цифрове мистецтво стає дедалі популярнішим. І музика не є виключенням.

Попри те, що на сьогоднішній день програмне забезпечення активно застосовуються для створення та обробки музичних композицій, які прийнято називати електронними, недостатньо вивченою залишається область їхньої автономної генерації електронно обчислювальними машинами із якомога найменшим втручанням людини. Звичайно, комп'ютери позбавлені таких важливих рис для цього, як креативності та художнього сприйняття. Але вже виконуються перші кроки для формалізація людських знань, аби заповнити дані пропуски та навчити програмне забезпечення виконувати ці функції.

В межах даної роботи за мету поставлено розглянути можливості функціональної мови програмування *Haskell* разом із популярною бібліотекою *Euterpea* для опису музичних фрагментів, проаналізувати та визначити особливості різних підходів штучного інтелекту та їхнього поєднання для генерації композицій, а також реалізувати власну програмну систему, яка створюватиме короткі музичні фрагменти без втручання користувача, засновуючись на досліджених методах.

Об'єктом дослідження є алгоритмічна композиція – створення музики за допомогою комп'ютерних алгоритмів, в той час як предметом дослідження обрано засоби мови *Haskell* та інтерфейс бібліотеки *Euterpea*.

Робота складається з двох розділів.

Перший розділ містить опис інструментів, використаних для реалізації програмної системи. Серед них бібліотека *Euterpea* та робота із псевдовипадковими величинами у *Haskell*.

Другий розділ починається із введення поняття алгоритмічної композиції та класифікації методів штучного інтелекту для даної задачі. Після того, наведено докладний опис створеної програмної системи.

Постановка задачі:

1. Виконати аналіз наданих засобів бібліотекою *Euterpea* для опису музичних композицій, а також розглянути роботу з псевдовипадковими величинами за допомогою монад та генераторів у *Haskell*.
2. Дослідити підходи штучного інтелекту для алгоритмічної композиції та способи їхнього поєднання.
3. Реалізувати власну програмну систему, яка генеруватиме короткі музичні фрагменти без втручання користувача.

РОЗДІЛ 1. ОПИС ІНСТРУМЕНТІВ

1.1. Бібліотека Euterpea

1.1.1. Загальні відомості

При дослідженні роботи з музикою в *Haskell*, була використана одна з популярних бібліотек для даної мови *Euterpea*, яка надає два рівня прикладного програмного інтерфейсу: сигнальний та нотний[1].

Перший з них оперує напряму зі звуковими сигналами, базуючись на їхньому дискретному представленні (неперервний аудіосигнал розкладається на послідовність дискретних одиниць, придатних для обробки ЕОМ), який є основою сучасної цифрової обробки звуку. На даному рівні сигнал характеризується трьома ключовими величинами: частотою, амплітудою та спектром, які впливають на висоту, гучність та тембр відповідно. У перелік можливих дій зі звуком входять інтерполяція, фільтрація, конвертація, осциляція, створення власних музичних інструментів, експорт в аудіофайл тощо.

Другий із них надає вищий рівень абстракції для роботи зі звуком. Елементарною одиницею на даному рівні є примітив, який може бути або нотою, або паузою. Таким чином, це дозволяє абстрагуватися від власне сигналів та працювати із музичними елементами. Такі примітиви можна поєднувати послідовно або паралельно, утворюючи композиції будь-якої складності. Крім того, можна транспонувати музичний твір, управляти музичними інструментами, темпом, та гучністю, виконувати експорт та імпорт *MIDI*-файлів.

В даній роботі було зосереджено увагу саме на нотному рівні прикладного програмного інтерфейсу для більш зручного створення музичних композицій за допомогою оперування нотами, а не сигналами. Наступні підрозділи описують головні компоненти даного рівня [2].

1.1.2. Ключові типи даних

1.1.2.1. Імпорт бібліотеки

Для початку роботи із *Euterpea* слід імпортувати бібліотеку на початку коду (рис. 1.1)

```
import Euterpea
```

Рисунок 1.1- імпорт Euterpea

1.1.2.2. Октава, нота, висота тону

Важливими типами даних є октава, нота та висота тону (рис. 1.2).

Octave - синонім до цілого типу *Int*, який позначає, до якої октави має належати нота.

PitchClass - тип даних, який позначає назву ноти у однойменних конструкторах. Суфікс *f* відповідає бемолю (від англ. *flat*), *s* - дієзу (від англ. *sharp*). Наприклад, конструктор *C* позначає ноту до, *Df* - ре бемоль, а *Gs* - соль дієз.

Pitch - синонім до пари назви ноти та номеру октави, який позначає висоту тону.

```
type Octave = Int
data PitchClass = C | Cs | Df | D | Ds | ... | B

type Pitch = (PitchClass, Octave)
```

Рисунок 1.2 - типи Octave, PitchClass, Pitch

1.1.2.3. Тривалість, примітив

Наступними вводяться поняття тривалості звуку та примітива (рис. 1.3).

Dur - синонім до типу раціональних чисел, позначає тривалість звучання певного звуку.

Primitive - узагальнений тип даних, який позначає елементарну одиницю музичної композиції. Він може містити у якості параметру *a*, наприклад *Pitch* або пару *(Pitch, Volume)* для налаштування гучності фрагментів. Конструктор *Note* відповідає за звучання певної висоти тону певної тривалості, в той час як *Rest* характеризується лише тривалістю та позначає паузу.

```
type Dur = Ratinonal

data Primitive a = Note Dur a | Rest Dur
```

Рисунок 1.3 – типи Dur, Primitive

1.1.2.4. Музичний фрагмент

Композиція усіх згаданих типів даних утворює музичний фрагмент, який визначається рекурсивно та має чотири конструктори (рис 1.4):

- *Prim* - примітив (нота чи пауза)
- *:+:* - два послідовних музичних фрагменти
- *:=:* - два паралельних музичних фрагменти
- *Modify* - модифікований музичний фрагмент (транспонований, зі зміненими інструментами, темпом тощо)

```
data Music a = Prim (Primitive a)
    | (Music a) :+: (Music a)
    | (Music a) :=: (Music a)
    | Modify Control (Music a)
```

Рисунок 1.4 - тип Music

1.1.2. Ключові функції та константи

1.1.2.1. Створення музичного фрагмента-примітива

При створенні музичного фрагменту, який складається із примітива, доводиться писати два конструктори. Оскільки це достатньо частий випадок, було введено дві функції, які скорочують даний запис (рис. 1.5).

Функція *note* ініціалізує музичний фрагмент, який складається зі звучання із заданими параметрами, в той час як *rest* - із паузи із заданою тривалістю.

```
note     :: Dur -> a -> Music a
note d p  = Prim (Note d p)

rest     :: Dur -> Music a
rest d    = Prim (Rest d)
```

Рисунок 1.5 – функції note, rest

1.1.2.2. Короткі назви тривалості

Для зручності подання тривалості звуку було введено короткі позначення (рис. 1.6), які відповідають найбільш вживаним величинам у нотному записі.

Деякими з них є: *wn* - ціла (від англ. *whole note*), *hn* - половинна (від англ. *half note*), *qn* - четвертна (від англ. *quarter note*) і так далі.

```
bn, wn, hn, qn, en, sn, tn, sfn :: Dur
bn  = 2
wn  = 1
hn  = 1 % 2
qn  = 1 % 4
en  = 1 % 8
sn  = 1 % 16
tn  = 1 % 32
sfn = 1 % 64
```

Рисунок 1.6 – значення тривалості

1.1.2.3. Створення музичного фрагмента із заданою нотою

Бібліотека має короткий спосіб створення музичних фрагментів, які складаються з ноти-примітива (рис. 1.7). Кожна з даних функцій має назву відповідної ноти, приймає два аргументи: октаву та тривалість, а повертає музичний фрагмент.

```
c, cs, df, d :: Octave -> Dur -> Music Pitch
c  o d = note d (C,  o)
cs o d = note d (Cs, o)
df o d = note d (Df, o)
d  o d = note d (D,  o)
...
```

Рисунок 1.7 – створення нот-примітивів

1.1.2.4. Композиція музичних фрагментів

Для послідовного та паралельного виконання двох фрагментів є відповідні конструктори типу даних *Music*, але часто необхідно виконати такі дії із багатьма фрагментами. Замість того, щоб кожного разу робити згортку списку фрагментів із заданим конструктором, можна скористатися готовими функціями бібліотеки (рис. 1.8): *line* для послідовної композиції, *chord* - для паралельної.

```
line, chord :: [Music a] -> Music a
line   = foldr (:+:) (rest 0)
chord  = foldr (:=:) (rest 0)
```

Рисунок 1.8 – функції line, chord

1.1.2.5. Відтворення музичних фрагментів

Після визначення музичного твору корисним є відтворення його за допомогою аудіоінтерфейсу комп’ютера. Для цього існує функція *play* (рис. 1.9), яка приймає на вхід музичний фрагмент та всередині монади *IO* (оскільки ця функція не є чистою, а має побічні ефекти - аудіосигнал) відтворює його.

```
play :: (ToMusic1 a, NFData a) => Music a -> IO()
```

Рисунок 1.9 – функція play

1.1.3. Приклад музичного фрагменту

У якості прикладу далі наведено фрагмент із композиції *“Happy Birthday”* авторства *Patty Hill*, використовуючи описані засоби бібліотеки *Euterpea*. Даний твір (рис. 1.10) складається з двох партій - мелодії та акордів.

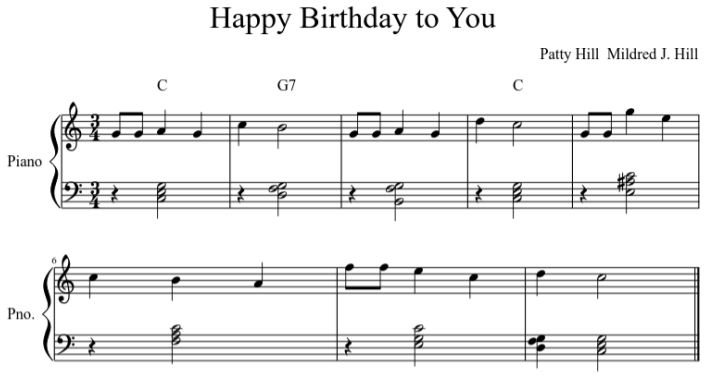

Рисунок 1.10 – нотний запис до фрагменту композиції "Happy Birthday", Patty Hill

1.1.3.1. Партія мелодії

При визначенні партії мелодії (рис. 1.11) деякі послідовності нот згруповані у фрази, які повторюються протягом композиції. Використовуються функції для швидкого створення фрагментів-примітивів на основі нот, а також функція *line* для послідовної композиції.

```
melody :: Music Pitch
melody = let    happyBirthday1 = line [g 4 en, g 4 en, a 4 qn, g 4 qn]
                toYou1         = line [c 5 qn, b 4 hn]
                toYou2         = line [d 5 qn, c 5 hn]
                happyBirthday2 = line [g 4 en, g 4 en, g 5 qn, e 5 qn]
                dearHaskell    = line [c 5 qn, b 4 qn, a 4 qn]
                happyBirthday3 = line [f 5 en, f 5 en, e 5 qn, c 5 qn]
         in     line [
                   happyBirthday1, toYou1,
                   happyBirthday1, toYou2,
                   happyBirthday2, dearHaskell,
                   happyBirthday3, toYou2
                ]
```

Рисунок 1.11 – визначення партії мелодії

1.1.3.2. Партія акордів

Структура партії акордів (рис. 1.12) дуже схожа на попередню. З однією відмінністю, що замість функції *line* використовується паралельна композиція *chord*. Крім того, тут наявні паузи завдяки використанню функції *rest*.

```
chords :: Music Pitch
chords = let    cChord1  = chord [c 3 hn, e 3 hn, g 3 hn]
                g7Chord1 = chord [d 3 hn, f 3 hn, g 3 hn]
                g7Chord2 = chord [b 2 hn, f 3 hn, g 3 hn]
                cChord2  = chord [e 3 hn, as 3 hn, c 4 hn]
                fChord   = chord [f 3 hn, a 3 hn, c 4 hn]
                cChord3  = chord [e 3 hn, g 3 hn, c 4 hn]
                g7Chord3 = chord [d 3 qn, f 3 qn, g 3 qn]
         in     line [
                   rest qn, cChord1, rest qn, g7Chord1,
                   rest qn, g7Chord2, rest qn, cChord1,
                   rest qn, cChord2, rest qn, fChord,
                   rest qn, cChord3, g7Chord3, cChord1
                ]
```

Рисунок 1.12 – визначення партії акордів

1.1.3.2. Фінальне відтворення

Маючи обидві партії, постає можливим відтворення заданого твору (рис.1.13). Для цього визначаємо функцію в монаді *IO* та використовуємо конструктор паралельної композиції, щоб партії відтворювалися одночасно.

```
demo :: IO ()
demo = play $ melody :=: chords
```

Рисунок 1.13 – функція demo

1.2. Робота із псевдовипадковими величинами

Оскільки всі функції в *Haskell* є чистими (крім тих, які використовують монаду *IO*), то реалізація роботи із псевдовипадковими величинами в ньому відрізняється від інших імперативних мов програмування.

Чисті функції повинні повертати той самий вихідний результат при тих самих вхідних даних, а також характеризуються відсутністю побічних ефектів. Саме тому неможливо у чистій функції створити псевдовипадкову подію, адже це може впливати на вихідні дані. Натомість, в *Haskell* використовуються генератори псевдовипадкових чисел [3].

1.2.1. Генератори псевдовипадкових чисел

Щоб розпочати роботу, необхідно виконати імпорт даного модуля (рис.1.14).

```
import System.Random
```

Рисунок 1.14 – імпорт System.Random

Генератор є екземпляром, який при виклику повертає псевдовипадкову величину, а також новий генератор. При повторному виклику генератор повертає той самий результат. Це забезпечує чистоту функцій, де він використовується. Таким чином, для безперервної генерації нових величин, слід замінювати поточний генератор новим.

Для роботи із генераторами псевдовипадкових чисел модуль надає декілька функцій (рис. 1.15).

Для створення генератора використовується функція *mkStdGen*, яка приймає цілочисельний параметр для визначення екземпляру. Функція *uniform* дозволяє отримати з генератора нове значення, яке належить діапазону значень заданого типа. За допомогою *uniformR* можна досягти схожого результату, але вказавши власний діапазон.

```
mkStdGen :: Int -> StdGen

uniform  :: (RandomGen g, Uniform a) => g -> (a, g)
uniformR :: (RandomGen g, UniformRange a) => (a, a) -> g -> (a, g)
```

Рисунок 1.15 – функції mkStdGen, uniform, uniformR

1.2.2. Використання монади State

Як зазначається вище, робота з генераторами псевдовипадкових чисел потребує постійного оновлення екземпляра. Було запропоновано використання монади *State* разом із ними для уникнення дублювання програмного коду та полегшення його сприйняття [4].

Спочатку слід виконати імпорт необхідного модуля (рис. 1.16).

```
import Control.Monad.State
```

Рисунок 1.16 – імпорт Control.Monad.State

Монада *State* є узагальненою та характеризується двома типами даних: станом та результатом. Протягом роботи в її межах відбувається опрацювання поточного стану, його оновлення новим значенням та продукування результату. Це дозволяють робити функції *get* та *put* (рис. 1.17), які відповідно виконують вибірку та оновлення стану.

```
get :: State s s
put :: s -> State s ()
```

Рисунок 1.17 – функції get, put

Таким чином, для роботи з генераторами псевдовипадкових величин корисно використовувати монаду *State* та тримати їх у якості стану в ній. Далі наведено приклад утиліти, яка дозволяє генерувати величини будь-якого типу (залежно від наданої функції) в межах даної монади (рис. 1.18).

```
randomElement :: (StdGen -> (a, StdGen)) -> State StdGen a
randomElement fun = do  generator <- get
                        let (el, nextGenerator) = fun generator
                        put $ nextGenerator
                        return el
```

Рисунок 1.18 – функція randomElement

Маючи даний інструмент, є можливість визначити конкретні функції для генерації псевдовипадкових подій, наприклад, цілих чисел в межах заданого діапазону (рис. 1.19).

```
randomInt :: Int -> Int -> State StdGen Int
randomInt minInt maxInt = randomElement (uniformR (minInt, maxInt))
```

Рисунок 1.19 – функція randomInt

1.2.3. Поєднання з монадою IO

Єдиною проблемою залишається встановлення параметра генератора, який повинен бути унікальним, якщо метою є генерація унікальних величин з кожним новим запуском програми. Для цього використовується функція *randomIO* всередині монади *IO* для генерації кожного разу псевдовипадкового цілого числа. Дане значення далі подається на вхід у створення генератора, забезпечуючи його унікальність. Також використовується функція *evalState* для розрахунку значення функції в межах монади *State* (рис. 1.20).

```
tryRandom :: (State StdGen a) -> IO a
tryRandom fun =
    do seed <- randomIO
       return $ evalState fun (mkStdGen seed)
```

Рисунок 1.20 - функція tryRandom

Перевіримо коректність даної утиліти за допомогою декількох послідовних викликів *randomInt* та переконаємося, що отримуємо кожного разу псевдовипадкове значення (рис. 1.21).

```
*Demo> tryRandom $ randomInt 0 100
62
*Demo> tryRandom $ randomInt 0 100
47
*Demo> tryRandom $ randomInt 0 100
76
*Demo> tryRandom $ randomInt 0 100
24
```

Рисунок 1.21 – демонстрація випадкових подій

РОЗДІЛ 2. АЛГОРИТМІЧНА КОМПОЗИЦІЯ

2.1. Опис підходів алгоритмічної композиції

2.1.1. Загальні відомості

Алгоритмічна композиція (від англ. *Algorithmic Composition*) - “...використання формальних процесів для створення музики з мінімальним втручанням людини, яке часто виконується на комп’ютері за допомогою різноманітних формалізмів, таких як генерації випадкових чисел, систем на основі правил, а також інших алгоритмів” (переклад з англ. - авторський)[5].

Іншими словами, це процес створення музики за допомогою комп’ютерних алгоритмів. Вплив людини може бути різним: від оцінки музичних творів, їхнього довершення, до його відсутності в абсолютно автономному написанні композицій. Чим меншим є втручання людини, тим більше вимог постає перед системою, так само як і проблем, які виникають протягом роботи; адже необхідно вбудовувати функціональність тих дій, відповідальність за яких можна було б перенести на користувача.

За час існування цієї задачі з’явилася велика кількість прикладів використання різних областей штучного інтелекту для її розв’язання[6], серед яких:

- граматики
- системі на основі знань
- ланцюги Маркова
- штучні нейронні мережі
- еволюційні методи
- самоподібність та клітинний автомат

В наступних підрозділах наявне коротке описання кожного з даних підходів.

2.1.2. Граматики

Формальні граматики (від англ. - *formal grammars*) - це інструмент опису формальної мови. Слова заданого алфавіту визначаються шляхом рекурсивного застосування певної множини правил, доки не залишаться лише термінальні символи. [6,с.520-526]

Оскільки існує теорія музики, яка містить певні правила гармонії (благозвучне поєднання нот, акордів, звукорядів), то її можна вважати граматикою, а музичну композицію - словом. Таким чином, задача зводиться до побудови певних слів, використовуючи алфавіт, який складається з нот і пауз, та дотримуючись заданих правил.

Важливою задачею є визначення граматики в системі, адже від неї залежить структура усіх породжених композицій. Вона може застосовуватися як на рівні всієї композиції (вказувати на її структуру, чергувати місцями куплет та приспів), так і на рівні окремих музичних фрагментів (власне вказувати послідовність нот і пауз для відтворення).

Важливим різновидом граматик є L-системи або системи Ліденмайєра, особливість яких полягає у тому, що на кожному етапі виконуються усі доступні перетворення замість одного. Тобто, усі нетермінальні вузли замінюються новою множиною символів одночасно. Вони є значно легшими для сприйняття, тому знайшли застосування в алгоритмічній композиції, у якості структури музичних творів, в якій кожен компонент (куплет, фраза, такт) можна описати рекурсивно як сукупність інших.

Крім вищезазначеного, граматики використовуються в парі з еволюційними методами в так званій граматичній еволюції (від англ. - *grammatical evolution*), аби обмежити простір пошуку індивідуумів до зазначеної структури.

2.1.3. Системи на основі знань

Різні системи, засновані на правилах та обмеженнях, прийнято називати системами на основі знань (від англ. - *rule-based systems*). Дана ідея в контексті

алгоритмічної композиції виникла досить природно: музиканти накопичили достатньо досвіду та знань щодо музичних творів, які настав час записати формально. Знаннями в даному контексті можуть бути різні ознаки композицій та їхній вплив на благозвучність (наприклад, мінімальна кількість інтервалів дисонансу). [6,с.526-536]

Незважаючи на те, що більша частина знань у більшості випадків є статичною, даний підхід активно застосовується з машинним навчанням, включно з ланцюгами Маркова та нейронними мережами. В такому разі сукупність нових знань створюється на основі аналізу великих наборів даних та виведення спільних ознак, часто окремо для різних музичних жанрів.

Іншим способом застосування даного підходу є еволюційні алгоритми, в яких введені знання застосовуються як фітнес-функція для згенерованих композицій, нараховуючи бонуси чи штрафи за виконання заданих ознак.

Найбільш яскравим прикладом використання систем на основі знань є задачі задоволення обмежень (від англ. - *constraint satisfaction problems* або *CSP*). Розв’язок таких задач полягає в тому, що алгоритм оптимізованим перебором будує можливі відповіді, на кожному етапі перевіряючи, чи не порушено жодне з вхідних обмежень.

Доволі відомим підходом є також судження, базоване на прецедентах (від англ. - *case-based reasoning*). В даній сукупності алгоритмів обов’язковим є наявність достатньої кількості прецедентів, або навчальної вибірки, які розподілені по задачам. Таким чином, при запиті на конкретну задачу система аналізує навчальну вибірку, виділяє найбільш доречну відповідь та адаптує її до поточного запиту, принагідно поповнюючи базу знань. Задачею в нашому контексті може бути музичний жанр, гармонія без мелодії тощо.

2.1.4. Ланцюги Маркова

Серед багатьох методів машинного навчання особливе місце в алгоритмічній композиції займають ланцюги Маркова (від англ. - *Markov chains*) - стохастичний дискретний процес, стан якого змінюється в межах скінченної

кількості, причому вибір на кожному етапі залежить лише від поточного стану (проте існують також розширення із так званою пам’яттю). [6,с.536-541]

Ланцюги Маркова можна інтерпретувати як зважений направлений граф, в якому вершини позначають стани, а ребра - переходи між ними із відповідними ймовірностями. Тим не менш, частіше використовують у якості позначення ймовірнісні матриці.

Станами, так само як і компоненти граматик, можна позначати як структурні одиниці композиції (куплети, приспіви тощо), так і конкретні музичні елементи (ноти та паузи). Щодо значень ймовірностей, то вони зазвичай розраховуються після навчання на вибірці даних, залежно від того, в якій частині від усіх випадків з даним станом, відбувався даний перехід (що також часто є специфічним для конкретних жанрів).

Завдяки невеликій кількості обчислень, ланцюги Маркова можуть використовуватися для музичної імпровізації, хоча зазвичай їх застосовують як чорновий варіант для доопрацювання композиторами, оскільки їхня генерація є достатньо випадковою.

2.1.5. Штучні нейронні мережі

Іншим популярним методом машинного навчання є штучні нейронні мережі (від англ. - *artificial neural networks* або *ANN*), які побудовані за принципом природних нейронних мереж та складаються з нейронів. Даний підхід у більшості випадків використовується у сукупності із навчальною вибіркою за принципом навчання з учителем (від англ. - *supervised learning*), який полягає в тому, що кожен прецедент має визначену реакцію на нього. [6,с.541-546]

Високою ступінню свободи характеризується інтерпретація вхідних та вихідних нейронів. Наприклад, можна для вхідних конфігурацій бажаної структури отримувати послідовність абсолютних величин висоти тону для створення нот, або для заданих мелодій певного жанру визначити гармонію (послідовності акордів тощо).

Окремо варто зазначити поєднання штучних нейронних мереж із еволюційними алгоритмами. Одним із застосувань є використання нейронних мереж як фітнес функції для індивідуумів, породжених еволюційним алгоритмом. В такому разі необхідно виконати етап тренування моделі, щоб вона могла виділяти ознаки в композиціях. Іншим способом, так званим інвертованим, є породження нейронних мереж еволюційним алгоритмом, при якому моделі оцінюються певними правилами та генерують музичні фрагменти.

2.1.6. Еволюційні методи

Наступною областю є еволюційні алгоритми (від англ. - *evolutionary algorithms*), які емулюють теорію Дарвіна природного відбору. Об'єкти, які генеруються алгоритмами, називається особами та об'єднуються в популяції (на початку згенеровану випадковим чином). Кожну ітерацію індивідууми оцінюються фітнес функцією та проходять відбір (від англ. - *selection*). До найкращих з них застосовуються оператори мутації (від англ. - *mutation*) та схрещування (від англ. - *recombination*), створюючи нову популяцію через репродукцію (від англ. - *reproduction*). [6,с.546-555]

В контексті алгоритмічної композиції, операторами можуть служити будь-які зміни музичних фрагментів. Проблемою постає вибір фітнес функції. Одним із простих способів є втручання користувача в оцінку композицій, який буде ставити у відповідність кількість балів. Дане рішення має кілька проблем. По-перше, важко оцінити витвори мистецтва за складною шкалою для побудови рейтингу особин. По-друге, результат роботи алгоритму буде залежати від суб'єктивного сприйняття користувача. Наостанок, часто за мету стоїть розробка автономного алгоритму, який здатен працювати без впливу людини.

Таким чином, виникає необхідність визначення автоматичної фітнес функції. У попередніх підрозділах було запропоновано кілька можливих рішень (системи на основі знань та штучні нейронні мережі). Загальним принципом обчислення балів є розрахунок суми балів за кожен критерій. Оскільки не існує універсальних критеріїв оцінки творчості, кожна система характеризується

специфічною фітнес функцією. Корисним правилом залишається використання правил гармонії, які можна подавати у вигляді граматики, правил тощо.

Іншим способом побудови фітнес функції є метрична відстань до даних прецедентів у навчальній вибірці. Щоправда, тут також постає питання вибору метрики як критерію схожості композицій.

Варто зазначити, що в даній роботі використовується термін еволюційні алгоритми замість генетичних, оскільки зазвичай при роботі з музичними творами важко представляти структуру у вигляді генотипу, тому використовуються інші способи інтерпретації (наприклад, граматики).

2.1.7. Самоподібність та клітинний автомат

Самоподібність (від англ. - *self-similarity*) - ознака, притаманна об’єктам, структура яких подібна до його частин (наприклад, фракталам). [6,с.555-557]

Дана характеристика притаманна багатьом класичним музичним творам, що дозволяє модифікувати наявні моделі та наблизити створені композиції до них. Зазвичай для генерації музичних фрагментів використовується декілька рівнів самоподібності із рідкими стохастичними змінами.

Клітинний автомат (від англ. - *cellular automata*) представляє собою n-вимірну сітку із елементарних одиниць - клітин, які характеризуються певним станом зі скінченної множини. Ці стани динамічно змінюються відповідно до заданих правил. [6,с.557-559]

Маючи множину клітин та їхніх станів, можна інтерпретувати їх як структурні елементи музичних фрагментів. Такий підхід є придатним також для імпровізації завдяки нескінченному обчисленню нових станів та відтворенню композиції. Модель можна навчати по правилам переходу станів клітин за допомогою штучних нейронних мереж та навчальної вибірки, а також ці правила можуть бути створені за допомогою еволюційних алгоритмів.

2.1.8. Висновки

Легко простежити, що для кожного із даних підходів відомо кілька способів його застосування з іншими. За допомогою поєднання кількох методів

із різних областей, постає можливість створити гібридну програмну систему більшої ефективності та генерувати музичні фрагменти вищої складності.

При реалізації алгоритмічної композиції в межах даної роботи було використано три з перелічених вище підходів: граматики, системи на основі знань та еволюційні алгоритми. Наступний підрозділ містить докладний опис створеної програмної системи.

2.2. Реалізація алгоритмічної композиції

2.2.1. Опис та структура проекту

Проект призначений для генерації коротких музичних фрагментів без втручання користувача, застосовуючи різні методи штучного інтелекту в контексті алгоритмічної композиції. Структура включає в себе п’ять модулів (рис. 2.1), які взаємодіють між собою.

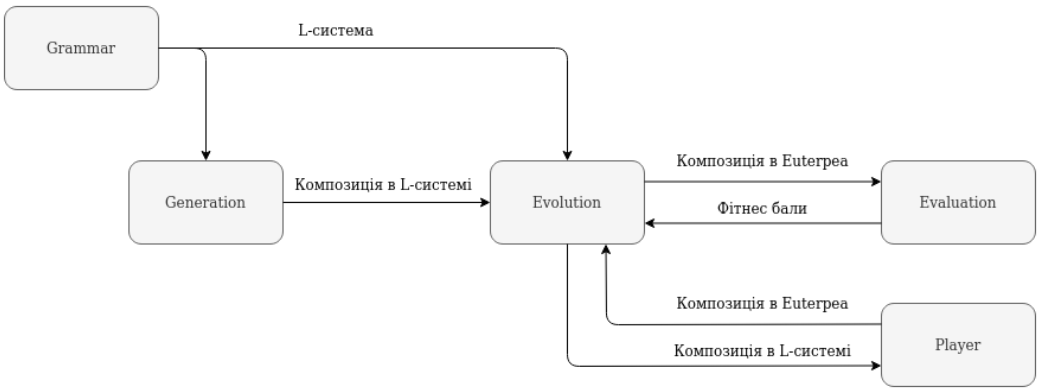

Рисунок 2.1 – Структура проекту

Модуль *Grammar* описує формальну граматику (L-систему) для створення композицій та надає зручний спосіб представлення музичних фрагментів і запитів користувача, які використовуються в інших частинах системи. Модуль *Generation* відповідає за породження випадкових композицій за даними запиту в межах визначеної граматики, які потім вдосконалюються. Модуль *Evolution* реалізовує еволюційний алгоритм, який випадковим чином ініціалізує початкову популяцію композицій та ітеративно застосовує оператори мутації та схрещування до них. Створені композиції модуль *Player* експортує у музичні фрагменти бібліотеки *Euterpea* та має можливість збереження фрагмента у *MIDI*-

файл. Функцію оцінки композицій надає модуль *Evaluation* - система на основі правил, яка приймає композиції та повертає відповідну суму балів.

Таким чином, граматика обмежує простір композицій до такого, в якому не допускаються грубі порушення гармонії. В той час як еволюційний процес шукає такі композиції, для яких система правил дає якомога більше балів, тобто які характеризуються більшою благозвучністю в межах заданого простору.

2.2.2. Модуль Grammar

2.2.2.1. Звукоряд та ритм

Будь-яка композиція характеризується звукорядом (від англ. - *Scale*), який визначає ноти, які можуть використовуватися в ній. Тип даних *ScaleMode* визначає тип звукоряду, для демонстрації якого наведено два конструктори: натуральні мажор (*IonianMode*) та мінор (*AeolianMode*). В той час як *Scale* визначає конкретний звукоряд із тонікою та типом (рис. 2.2).

```
data ScaleMode = IonianMode | AeolianMode deriving Show
data Scale = Scale PitchClass ScaleMode deriving Show
```

Рисунок 2.2 – типи ScaleMode, Scale

Кожен звукоряд - це певна послідовність інтервалів, які починають відлік від ноти-тоніки композиції. Далі наведена функція, яка кожному із заданих типів звукорядів ставить у відповідність список ступенів (рис. 2.3).

```
modeIntervals :: ScaleMode -> [Int]
modeIntervals IonianMode = [0, 2, 2, 1, 2, 2, 2, 1]
modeIntervals AeolianMode = [0, 2, 1, 2, 2, 1, 2, 2]
```

Рисунок 2.3 – функція modeIntervals

Музичний розмір характеризує ритм композиції та складається з двох чисел - довжина тактової долі, а також їхня кількість в такті. За дану сутність відповідає тип *Rhythm* (рис. 2.4).

```
data Rhythm = Rhythm Integer Integer deriving Show
```

Рисунок 2.4 - тип Rhythm

2.2.2.2. Акорд та прогресія акордів

Одним із ключових понять теорії музики та гармонії є акорд - сукупність нот, які відтворюються одночасно. Так само як і зі звукорядом, в граматиці визначено типи *ChordType* та *Chord*, які позначають відповідно тип акорду (наприклад, мінорний та мажорний) та конкретний акорд із тонікою (рис. 2.5). Тут в якості тоніки вказується не нота, а ступінь звукоряду, аби дозволити побудову акордів лише від тих нот, які належать йому (*ScaleStep* - синонім до цілого числа).

```
data ChordType = MinorChord | MajorChord deriving Show
data Chord = Chord ScaleStep ChordType deriving Show
```

Рисунок 2.5 – типи ChordType, Chord

Також визначено функцію, яка для кожного типу акорду повертає список інтервалів, які відраховуються від тоніки (рис. 2.6).

```
chordsIntervals :: ChordType -> [Int]
chordsIntervals MinorChord = [0, 3, 4]
chordsIntervals MajorChord = [0, 4, 3]
```

Рисунок 2.6 – функція chordsIntervals

Прогресією акордів називають певну послідовність акордів, яка повторюється протягом композиції. В граматиці даної системи кожен акорд триває однакову кількість часу, яка є параметром прогресії (рис. 2.7).

```
data ChordProgression = ChordProgression [Chord] Dur deriving Show
```

Рисунок 2.7 – тип ChordProgression

2.2.2.3. Розподіл

В межах проекту розподілом (*Partition*) називається структура певного проміжку часу, який займає один акорд. Він складається зі списку частин (*Part*), кожна з яких триває визначену кількість долей такту (сума завжди стала та визначається ритмом). Неподільні частини (*SolidPart*) містять один елемент, що триває визначену кількість долей такту, в той час як подільні (*DividablePart*) можуть рекурсивно ділитися на два рівних відрізки, поки в кожному

термінальному вузлі не буде заданого елементу (рис. 2.8). Всередині можуть зберігатися будь-які дані, адже тип є узагальненим.

```
data Partition a =
    Partition [Part a]
        deriving Show
data Part a =
    SolidPart Integer a |
    DividablePart (DividableElement a)
        deriving Show
data DividableElement a =
    DoubleElement (DividableElement a) (DividableElement a) |
    SingleElement a
        deriving Show
```

Рисунок 2.8 – типи Partition, Part, DividableElement

2.2.2.4. Партія та стратегії

Програмна система пропонує три стратегії виконання партії: акордів, мелодії та арпеджіо. Кожен з них має в якості параметра мінімальну тривалість елемента (аби уникнути нескінченно глибоких розподілів), бульовий параметр, який позначає, чи можна елементам об’єднуватися в довгі неподільні частини, а також самі дані - розподіли елементів та список інверсій, за необхідності. Інверсією називають модифікацію акорду, коли одна з крайніх нот зміщується на октаву, тобто її можна представити у вигляді цілого числа - кількості таких зміщень (рис. 2.9).

```
type Inversion = Int

data PartStrategy =
    ArpeggioStrategy Dur Bool (Partition ArpeggioElement) [Inversion]  |
    ChordsStrategy Dur Bool (Partition ChordElement) [Inversion]  |
    MelodyStrategy Dur Bool [[Partition MelodyElement]]
        deriving Show
```

Рисунок 2.9 – типи Inversion, PartStrategy

В якості прикладу наведено визначення типу даних елементу мелодії (*MelodyElement*), який позначає або паузу, або відтворення даної ступені звукоряду (рис. 2.10). Такий вибір замість вказання назви ноти було зроблено, щоб унеможливити використання тих нот, які не входять до звукоряду [7].

```
data MelodyElement =
    MelodyNote ScaleStep |
    MelodyRest
        deriving Show
```

Рисунок 2.10 – тип MelodyElement

2.2.2.5. Композиція

Найбільш загальним типом даних є музична композиція (*Composition*), яка містить у собі кількість відтворювань прогресії акордів, звукоряд, ритм, саму прогресію акордів, а також список партій. У свою чергу, кожна з партій характеризується даними про стратегію, октаву відтворення, музичний інструмент та гучність (рис. 2.11).

```
type PartMetaData = (PartStrategy, Octave, InstrumentName, Volume)
data Composition = Composition Int Scale Rhythm ChordProgression [PartMetaData] deriving Show
```

Рисунок 2.11 – типи PartMetaData, Composition

2.2.3. Модуль Generation

2.2.3.1. Генерація елементів партії

Найменшими структурними одиницями композиції є елементи партії. Для кожної стратегії визначено власні функції всередині монади *State* для генерації відповідних елементів. Далі наведено приклад для стратегії мелодії (рис. 2.12). Варто зазначити про використання функції *randomFromList*, яка є утилітою для вибору елементу зі списку із даним розподілом. Щодо номеру ступені звукоряду, то в даному прикладі використовується генерація цілого числа без обмежень, від якого згодом буде вирахувано модуль від ділення на кількість ступеней в конкретному звукоряді, адже їхня кількість може варіюватися.

```
generateMelodyRest :: State StdGen MelodyElement
generateMelodyRest = do return MelodyRest

generateMelodyNote :: State StdGen MelodyElement
generateMelodyNote = do step <- randomAnyInt
                        return $ MelodyNote step

generateMelodyElement :: State StdGen MelodyElement
generateMelodyElement =
    do choice <- randomFromList [(generateMelodyRest, 0.4), (generateMelodyNote, 0.6)]
       choice
```

Рисунок 2.12 – функції generateMelodyRest, generateMelodyNote, generateMelodyElement

Таким чином, правила заміни обираються випадково, що свідчить про те, що граматика є стохастичною [8].

2.2.3.2. Генерація розподілу

Випадкова генерація розподілу виконується для кожної партії окремо. В якості одного з аргументів передається функція генерації відповідних елементів

(як-от вищезгадана *generateMelodyElement*), яка викликається для заповнення термінальних вузлів. В якості прикладу наведено деякі з функцій побудови частин розподілу (рис. 2.13), в яких одне й те саме правило може застосовуватися рекурсивно кілька разів [9]. Варто додати, що в *generateDividableElement* здійснюється вибір між розгалуженням та термінальним вузлом, згідно з деяким розподілом, принагідно перевіряючи, чи не порушена мінімальна тривалість елементів.

```
generateDividableElement :: Dur -> Dur -> State StdGen a -> State StdGen (DividableElement a)

generateDoubleElement :: Dur -> Dur -> State StdGen a -> State StdGen (DividableElement a)
generateDoubleElement minDur currDur fun =
    do  firstEl <- generateDividableElement minDur (currDur / 2) fun
        secondEl <- generateDividableElement minDur (currDur / 2) fun
        return $ DoubleElement firstEl secondEl

generateSingleElement :: State StdGen a -> State StdGen (DividableElement a)
generateSingleElement fun = do res <- fun
                               return $ SingleElement res
```

Рисунок 2.13 – функції generateDividableElement, generateDoubleElement, generateSingleElement

2.2.3.3. Генерація композиції

Власне композиція генерується за вхідним запитом, що містить дані для опису бажаної структури. Запити на партії служать параметром для їхньої генерації, поки всі інші компоненти надходять напряму в конструктор композиції (рис. 2.14).

```
generateComposition :: CompositionPlaceholder -> State StdGen Composition
generateComposition placeholder@(CompositionPlaceholder count chordProgression _ scale rhythm) =
    do  parts <- generateCompositionParts placeholder
        return $ Composition count scale rhythm chordProgression parts
```

Рисунок 2.14 – функція generateComposition

2.2.4. Модуль Evolution

2.2.4.1. Налаштування параметрів

Для роботи ітеративного процесу еволюційного алгоритму варто визначити необхідний набір величин, серед яких кількість особин в популяції, ітерацій та лідерів (тих індивідуумів із найбільшим значенням фітнес функції, які породжують наступну популяцію) (рис. 2.15).

```
individualsAmount :: Int
iterationsAmount :: Int
leadersAmount :: Int
individualsAmount = 40
iterationsAmount = 100
leadersAmount = 10
```

Рисунок 2.15 – конфігурація еволюційного алгоритму

2.2.4.2. Загальна схема еволюції

Процес еволюції складається з кількох етапів. Усі вони використовують монаду *IO* через необхідність створювати нові генератори для внутрішніх дій, а також для відтворення за зберігання композицій (рис. 2.16).

Точкою входу є функція *runEvolution*. Спочатку ініціалізується початкова випадкова популяція функцією *initializePopulation*, яка приймає запит на структуру композиції, включно зі списком партій. Потім алгоритм ітерується всередині *iteratePopulations* задану кількість разів. Найкращий фрагмент конвертується, зберігається та відтворюється (відповідні функції розглядаються далі).

```
initializePopulation :: CompositionPlaceholder -> IO [Composition]
iteratePopulations :: CompositionPlaceholder -> [Composition] -> Int -> IO Composition

runEvolution :: CompositionPlaceholder -> IO ()
runEvolution placeholder =
    do initPopulation <- initializePopulation placeholder
       bestComposition <- iteratePopulations placeholder initPopulation iterationsAmount
       let music = playComposition bestComposition
       saveMidi music
       play music
```

Рисунок 2.16 – функції initializePopulation, iteratePopulations, runEvolution

2.2.4.3. Оператори мутації та схрещування

Для зміни музичних композицій всередині алгоритму використовуються два типи операторів (рис. 2.17). Схрещування полягає в обміні партій одного типу між двома композиціями. Мутації передбачено декількох типів, які застосовуються до кожного розподілу:

- заміна термінального вузла розгалуженням із рекурсивною генерацією
- заміна розгалуження згенерованим термінальним елементом
- повторна генерація елементів у термінальних вузлах
- рекурсивна мутація частин розгалуження
- дублювання однієї з частин розгалуження

- об’єднання та розділення кореневих частин

Усі з перерахованих операторів застосовуються із визначеною ймовірністю та обираються відповідно до певного імовірнісного розподілу, залежно від типу поточного вузла.

```
crossOverPopulation :: [Composition] -> State StdGen [Composition]
mutateCompositionsList :: [Composition] -> State StdGen [Composition]
```

Рисунок 2.17 – функції crossOverPopulation, mutateCompositionList

Деякі з правил мутацій засновані на попередніх роботах [10].

Застосування операторів, які модифікують значення вузла залежно від сусідніх (в даному випадку дублювання частин розгалуження), було вирішено перенести саме в модуль еволюційного алгоритму, на відміну від пропозиції зробити їх частиною граматики [11]. Зроблено це для того, аби граматика залишалась достатньо простою, а також аби після застосування операторів було можливим подальше застосування мутацій.

2.2.5. Модуль Player

2.2.5.1. Відтворення композиції

Для відтворення музичного фрагменту необхідно конвертувати його із представлення в L-системі у структури бібліотеки *Euterpea*. Для цього ми виконуємо дану операцію для кожної партії, залежно від стратегії, та застосовуємо паралельну композицію, адже вони мають звучати одночасно (рис.2.18).

```
playComposition :: Composition -> Music (Pitch, Volume)
playComposition (Composition count scale rhythm chordProgression strategies) =
    chord $ map (playPart count scale rhythm chordProgression) strategies
```

Рисунок 2.18 - функція playComposition

Окремо варто зазначити про відтворення частин на основі розподілу. З даною ціллю визначено функцію *playElementWrapper*, яка приймає відтворювач елементів, з яких складається розподіл (для кожної стратегії свій), та застосовує його до термінальних вузлів, а для розгалужень рекурсивно викликає себе (рис.2.19).

```
playElementWrapper :: PlayElementData -> b -> DividableElement a -> (PlayElementData -> b -> a -> Music Pitch) -> Music Pitch
playElementWrapper playData additional (SingleElement element) player =
    player playData additional element
playElementWrapper (PlayElementData scale octave duration) additional (DoubleElement element1 element2) player =
    (playElementWrapper (PlayElementData scale octave (duration / 2)) additional element1 player)
    :+:
    (playElementWrapper (PlayElementData scale octave (duration / 2)) additional element2 player)
```

Рисунок 2.19 - функція playElementWrapper

2.2.5.2. Збереження композиції у вигляді MIDI-файлу

Бібліотека *Euterpea* надає можливість експорту музичного фрагменту у файл формату *MIDI*. Для цього необхідно всередині монади *IO* вказати ім’я шлях до місця призначення та передати об’єкт *Music*. В межах даного модуля в якості назви директорії для файлу використовується значення поточного часу для уникнення проблем із дублюванням імен (рис. 2.20).

```
saveMidi :: Music (Pitch, Volume) -> IO ()
saveMidi music = do currTime <- getCurrentTime
                    let currTimeStr = show currTime
                    createDirectoryIfMissing True $ "../out/" ++ currTimeStr
                    let fileName = "../out/" ++ currTimeStr ++ "/midi.midi"
                    writeMidi fileName music
```

Рисунок 2.20 – функція saveMidi

2.2.6. Модуль Evaluation

2.2.6.1. Вартість інтервалів

Важливою частиною благозвучності композиції є інтервали, з яких вона складається. Засновуючись на цьому, даний модуль містить визначення балів за кожен тип інтервалу, які призначаються протягом оцінювання: різні види консонансу та дисонанс (рис. 2.21).

```
perfectConsonanseScore :: Double
dissonanceScore :: Double
...
perfectConsonanseScore = 10
dissonanceScore = -50
...

intervalToScore :: Int -> Double
intervalToScore diff =
    fromMaybe (dissonanceScore) $ lookup diff [
        (0, perfectConsonanseScore),
        (1, dissonanceScore),
        (2, dissonanceScore),
        ...
    ]
```

Рисунок 2.21 – вартість музичних інтервалів

2.2.6.2. Оцінка музичних фрагментів

Оцінка композицій в межах даної системи складається з двох частин (рис.2.22). Критерії були запропоновані в попередніх роботах [12, 13].

Перша з частин, оцінка партії, приймає окремо кожну партію фрагменту та нараховує бали по таким критеріям:

- благозвучність інтервалів між нотами, які знаходяться поруч
- різноманіття абсолютних висот тону по всім нотам
- не надто велика відстань між найнижчою і найвищою нотами
- якомога більше нот, які належать поточному акорду

Другою є оцінка паралельного звучання, яка приймає усі партії та в кожен момент часу виділяє такий критерій:

- благозвучність інтервалів між нотами, які звучать одночасно

```
evaluatePart :: Scale -> ChordProgression -> Music Pitch -> Double
evaluateParallel :: [Music Pitch] -> Double

evaluateMusic :: Scale -> ChordProgression -> Music Pitch -> Double
```

Рисунок 2.22 – функції evaluatePart, evaluateParallel, evaluateMusic

Висновки

Протягом виконання даної роботи було досягнуто поставленої мети.

Проаналізовано можливості бібліотеки *Euterpea* та встановлено, що за допомогою неї можна описувати музичні композиції будь-якої складності. Розглянуто способи роботи із псевдовипадковими величинами та показано доцільність використання монад *State* та *IO* разом із генераторами.

Досліджено та класифіковано такі методи штучного інтелекту для алгоритмічної композиції: граматики, системи на основі знань, ланцюги Маркова, штучні нейронні мережі, еволюційні алгоритмі, самоподібність та клітинний автомат. Продемонстровано способи їхнього поєднання для створення гібридних програмних систем.

Реалізовано власний проект для автономної генерації коротких музичних фрагментів без втручання людини, який складається з кількох модулів і використовує граматики, еволюційні алгоритми та системи на основі знань. Серед можливих варіантів розвитку проекту є:

- розширення граматики новими структурами (стратегіями, типами акордів та звукорядів)
- поповнення списку операторів мутації та схрещування
- збільшення кількості ознак для оцінки композицій
- застосування методів машинного навчання для генерації музичних фрагментів конкретних жанрів

Список використаних джерел

[1] Hudak P. The Haskell School of Music / Paul Hudak. – New Haven: Cambridge University Press, 2018. – 398 с. – (1st edition).

[2] Euterpea.Music Documentation [Електронний ресурс] – Режим доступу до ресурсу: http://hackage.haskell.org/package/Euterpea-2.0.4/docs/Euterpea-Music.html.

[3] System.Random Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://hackage.haskell.org/package/random-1.2.0/docs/System-Random.html#t:StdGen.

[4] O'Sullivan B. Real World Haskell / B. O'Sullivan, J. Goerzen, D. Stewart. – O'Reilly Media, 2008. – 714 с. – (1st edition).

[5] Alpern A. Techniques for Algorithmic Composition of Music / Adam Alpern. – 1995.

[6] Fernandez J. AI Methods in Algorithmic Composition: A Comprehensive Survey / J. Fernandez, F. Vico. – 2014.

[7] Fox C. Genetic Hierarchical Music Structures / Charles Fox. – 2006.

[8] McCormack J. Grammar-Based music composition / Jon McCormack. – 1996.

[9] Quick D. Grammar-based automated music composition in Haskell / D. Quick, P. Hudak. – 2013.

[10] Donnelly P. Evolving Four-Part Harmony Using Genetic Algorithms / P. Donnelly, J. Sheppard. – 2011.

[11] Reddin J. Elevated Pitch: Automated Grammatical Evolution of Short Compositions / J. Reddin, J. Mcdermott, M. O'Neill. – 2009.

[12] Towards melodic extension using genetic algorithms / M.Towsey, A. Brown, S. Wright, S. Diederich. – 2001.

[13] Papadopoulos G. A Genetic Algorithm for the Generation of Jazz Melodies / G. Papadopoulos, G. Wiggins. – 2000.