캡스톤디자인I (CSE4186) 중간고사 리포트

전공: 컴퓨터공학과 학년: 3학년 학번: 20211547 이름: 신지원

1.

문제 이해:

Sequence 의 여러 단계를 처리할 때는 전체 단계의 결과를 요청할 때 실제 연산이 이루어지기 때문에 lazy 하게 처리되는 특징이 있다. sequence 가 수행되며 element 를 독립적으로 처리하는 대표적인 방식이 map(), filter() 이다. 이때, sequence 수행 동작이 lazy 하면서 다른 sequence 를 리턴 한다면 그것은 intermediate 라 하고 다른 동작은 terminal 이라 한다.

A.

map function

```
public fun <T, R> Sequence<T>.map(transform: (T) -> R): Sequence<R> {
    return TransformingSequence(this, transform)
}
```

TransformingSequence class

```
class TransformingSequence<T, R>(
    private val sequence: Sequence<T>,
    private val transformer: (T) -> R
) : Sequence<R> {
    override fun iterator(): Iterator<R> = object : Iterator<R> {
        private val iterator = sequence.iterator()

        override fun hasNext() = iterator.hasNext()

        override fun next() = transformer(iterator.next())
    }
}
```

Map function 은 Sequence와 변환 함수를 TransformingSequence에 전달하여, 각 element 를 필요할 때 변환하도록 한다. 문제 이해에서 언급하였던 것처럼, map 함수는 특정 state 를 요청하는 것이 아닌 element 를 독립적으로 처리한다. 나아가 이 함수는 intermediate operation 으로, 반환된 Sequence는 요소에 대한 실제 연산을 실행하지 않고, 그 연산을 lazy 상태로 유지한다.

코드를 구체적으로 살펴보자면, map function 은 Sequence<T> 타입의 데이터를 받아 각 요소에 특정 변환(transform)을 적용하고, 그 결과를 Sequence<R>로 반환하는 확장 함수이다. map 함수

는 TransformingSequence 클래스의 인스턴스를 생성하여 반환하며, TransformingSequence 객체 생성 시, 원본 Sequence (this)와 변환 함수 (transform)가 전달된다.

TransformingSequence class 는 확장 함수 map 의 구체적인 지연 계산을 수행한다. map 함수는 요소에 대한 실제 연산을 수행하지 않으며 TransformingSequence class 에서 수행하게 되는 것이다.

sequence 는 변환해야 할 Sequence 의 객체를 의미하며, transformer 는 T에서 R 타입으로 변환하는 함수다. iterator() 에서는 element 들을 순차적으로 돌며 래핑하고 변환하는 역할을 수행한다. 사용자가 변환된 Sequence<R>에서 요소를 요구할 때 (ex, toList, first, forEach 등), TransformingSequence의 iterator() 메서드가 호출된다. 이 메서드는 새로운 Iterator<R>를 생성하는데 내부 Iterator<R>는 원본 Iterator<T>로부터 요소를 하나씩 가져와 transformer 함수를 적용하여 요소를 새로운 타입 R로 변환한다.

B.

first function

```
fun <T> Sequence<T>.first(): T {
    val iterator = iterator()
    if (!iterator.hasNext()) throw NoSuchElementException("Sequence is empty.")
    return iterator.next()
}

fun <T> Sequence<T>.first(predicate: (T) -> Boolean): T {
    val iterator = iterator()
    while (iterator.hasNext()) {
        val element = iterator.next()
        if (predicate(element)) return element
    }
    throw NoSuchElementException("No element matching predicate was found.")
}
```

first 함수는 Sequence의 첫 번째 요소를 반환하는 terminal operation 이다. 이 함수는 Sequence를 즉각 평가하며, 첫 번째 요소를 찾는 즉시 평가를 중단한다. 주어진 조건이 있다면 각 요소를 평가하고 조건을 만족하는 첫 번째 요소를 반환하며, 조건을 만족하는 요소가 없을 경우 예외를 발생시킨다.

Intermediate Operation (map): map은 결과를 즉시 생성하지 않고, 각 요소에 변환 함수를 적용하여 새로운 Sequence를 생성한다. 여기서는 terminal operation이 호출될 때까지 연산을 수행하지 않는다.

Terminal Operation (first): first 함수는 Sequence에서 첫 번째 요소를 그 즉시 평가하고 결과값을 반환한다. 호출되는 순간 Sequence에 대한 연산이 시작되며, 이는 필요한 데이터만 계산하여 효

율적인 데이터 처리를 가능하게 한다.

즉, Sequence에서의 lazy evaluation은 intermediate operation을 통해 정의된 연산을 즉시 수행하지 않고 연산의 정의만을 저장하는 방식으로 구현된다. 이렇게 연산을 지연시키는 방식은 메모리 사용을 최적화하고, 필요하지 않은 계산을 피하여 성능을 개선한다. 이후 Terminal operation이 호출된다면, 그때까지 쌓인 모든 지연 연산들을 필요한 만큼만 수행한다. 예를 들어, first()나 count() 같은 연산은 전체를 연산하지 않고 일부 요소에 대해서만 연산을 수행할 수도 있다. 이는 필요한 데이터만 처리함으로써 자원 사용을 크게 줄일 수 있다. 따라서 각각의 수행 방법은 모두 각자의 방법으로 성능 계선과 효율성을 위한 것이다.

2.

문제이해.

```
public interface List<out E> : Collection<E> {
    public operator fun get(index: Int): E
}

public interface MutableList<E> : List<E>, MutableCollection<E> {
    public fun add(element: E): Boolean
    public fun add(index: Int, element: E)
}
```

A.

불변성은 컴파일 타임 에러를 잡아주고 런타임에 에러를 내지 않는 안전한 방법이다. 하지만 불필요한 불변성 때문에 문제가 발생할 수 있는데, 이때 out 키워드를 통하여 공변성으로 변환할 수 있다.

위 코드 List에서 E는 앞서 언급한 out 키워드로 선언되어 있다. 이는 쉽게 이야기하면, List 가 Read 가 가능하고 Write 는 불가능하다는 것을 의미한다. 즉, List<E>는 공변성을 가지며, List<String>은 List<Any>의 하위 타입이 될 수 있다.

하지만 MutableList에서 E에는 variance annotation 이 없다. 이는 MutableList 가 요소 E를 읽고 쓸 수 있다는 것을 의미한다. 따라서 MutableList는 불공변성을 가지며, MutableList<String>은 MutableList<Any>의 하위 타입이 아니다.

이러한 차이를 가지는 이유는 List는 변경할 필요가 없기 때문에 읽을 수만 있으면 되기 때문이다. 따라서 out을 사용하여 더 일반적인 타입의 List로 할당할 수 있다. 하지만 MutableList는 요소를 추가하거나 변경할 수 있기 때문에 요소 타입에 대한 엄격한 일치가 필요하여 공변을 허용하지 않는 것이다.

B.

```
// 1
List<T>.contains(element: T): Boolean
// 2
List<T>.indexOf(element: T): Int
// 3
List<T>.lastIndexOf(element: E): Int
```

이 메소드들은 T 타입의 요소를 인수로 받고 있으며 type parameter variance 에 위배되고 있다. T 타입을 읽기만 하는 것이 아니라 쓰기를 요청하기 때문이다. 이에 대해 Compile Time Error를 회피를 위해 사용된 방법이 있다. 실제로 위 예시들은 type parameter variance 를 해치지 않고 있다. 그 이유는 위 함수들은 읽기만의 용도이며 타입 파라미터를 사용하고 있기 때문이다.

다시 말하자면, Contains, indexOf, lastIndexOf 함수들은 읽기 용도의 함수기 때문에 쓰기(소비)가 수행될 일이 없기에 안전함을 알 수 있다. 또한 제네릭 타입 사용 시 타입 안전성을 철저히 검사하여, 타입 불일치가 발생할 경우 컴파일 에러를 발생시키는데, 위 예시의 경우 타입 파라미터 T는 리스트의 구조 변경 없이 검사 목적으로만 사용되기 때문에 문제가 되지 않을 수 있다. 즉, T를 받아들이지만, 이들은 컬렉션의 상태를 변경하지 않으므로 안전하다는 것이다.

C.

B에서 제시한 메소드들은 제네릭 타입 T를 쓰면서(소비하면서) 사용하는 것처럼 보이지만, 실제로는 타입을 변경하거나 리스트의 상태를 변경하는 것 없이 읽기의 역할만을 수행한다.

contains(element: T): Boolean:

이 메소드는 리스트에서 주어진 요소 element가 존재하는지 여부만 확인하고, 리스트의 상태를 변경하지 않는다. T 타입의 요소를 받아들이지만, 이는 리스트 내부의 요소와 비교하는 용도로만 사용된다.

indexOf(element: T): Int:

indexOf 메소드는 리스트에서 주어진 요소 element의 인덱스를 찾는다. 이 연산 역시 리스트의 상태를 변경하지 않으며, 요소가 리스트에 존재하는 위치만을 반환한다.

이 메소드들은 리스트의 내부 데이터를 변경하거나 요소를 추가하지 않기 때문에, type-safe하다. 리스트의 불변성을 유지하면서도, 요소의 검색과 같은 읽기 전용 작업을 수행할 수 있다

lastIndexOf(element: T): Int

lastIndexOf 메소드는 주어진 요소 element가 리스트 내에서 마지막으로 나타나는 위치의 인덱스를 반환한다. lastIndexOf 또한 리스트의 상태를 변경하지 않으며, 요소의 추가나 수정 없이 순수하게 조회만을 수행한다. 리스트를 끝에서부터 역순으로 순회하면서 요소를 비교하고, 요소의 위치 정보만을 반환한다.

3.

A.

also 함수는 T 의 확장함수로 수신 객체가 암시적으로 제공되며, 수신 객체 지정 람다 에 매개변수 T 로 코드 블록 내에 명시적으로 전달되고, 코드 블록 내에 전달된 수신객체를 그대로 다시 반환한다.

따라서 swap 이 가능한 이유는 다음과 같은 순서로 설명할 수 있다. 먼저, also 함수의 람다 내부에서 b의 현재 값(2)이 also 함수의 리턴 값으로 지정된다. 그 후 { b = a } 람다내부가 실행되어 b 변수에 a의 값(1)이 할당된다. 마지막으로 also로부터 반환된 값(2)이 a에 할당된다.

B.

```
// apply
a = b.apply { b = a }

// let
a = b.let { temp -> b = a; temp }

// run
a = b.run { val temp = this; b = a; temp }
```

apply 함수는 수신 객체 람다 내부에서 수신 객체의 함수를 사용하지 않고 수신 객체 자신을 다시 반환한다. 수신 객체 의 프로퍼티 만을 사용하는 대표적인 경우가 객체의 초기화 이며, 이러한 곳에 apply 를 사용한다.

위 코드에서 apply는 호출된 객체 b를 수정한 후 그 객체를 반환하며, b의 값을 a로 설정하고 수정된 b를 a에 할당한다.

let 함수는 지정된 값이 null 이 아닌 경우에 코드를 실행해야 하는 경우, Nullable 객체를 다른 Nullable 객체로 변환하는 경우, 단일 지역 변수의 범위를 제한 하는 경우에 주로 사용한다.

위에서 let 함수는 호출된 객체 b를 람다 블록의 인자로 넘겨주고, 람다 블록의 결과를 반환한다. 람다 내에서 b에 a를 할당하고, 람다의 결과로 원래의 b 값을 a에 할당한다.

어떤 값을 계산할 필요가 있거나 여러 개의 지역 변수의 범위를 제한하는 경우에 run 을 사용한

다. 매개 변수로 전달된 명시적 수신객체를 암시적 수신 객체로 변환할 때 run() 을 사용할 수 있다.

run은 apply와 유사하게 수신 객체 b의 컨텍스트 내에서 람다를 실행하고, 람다의 결과를 반환한다. 제공한 위 코드에서 임시 변수 temp에 this를 저장하고, b에 a의 값을 할당한 후, temp를 반환하여 a에 할당한다.

C.

```
// also
inline fun <T> T.also(block: (T) -> Unit): T {
    block(this)
    return this
}

// Usecase
val numbers = mutableListOf(1, 2, 3).also {
    println("Initial list: $it")
}.map { it * 2 }.also {
    println("Doubled list: $it")
}
```

also 함수는 호출하는 객체를 블록에 전달하고, 블록을 실행한 후 객체를 반환한다. 블록의 매개변수는 호출된 객체(여기서는 this) 와 동일하다.

Also 함수는 객체를 변경하는 과정을 로깅하거나 디버깅할 때 유용하게 사용된다. 객체의 상태 변화를 추적하면서도, 해당 객체의 흐름을 방해하지 않고 계속해서 체이닝할 수 있다.

```
// apply
inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}

// Usecase
val JiwonView = TextView(context).apply {
    text = "I'm Jiwon"
    textSize = 100f
    setTextColor(Color.PINK)
}
```

apply 함수는 호출된 객체에 대해 블록을 실행하고, 객체 자신을 반환한다. 이때 블록은 객체의 context 내에서 실행되어, 객체의 멤버에 직접 접근할 수 있는 특징을 갖는다.

apply 함수는 객체의 여러 속성을 초기화할 때 주로 사용된다. 이는 특히 객체 생성과 동시에 여러 설정을 한 번에 수행해야 할 때 유용하다. 위 코드에서 apply는 TextView 객체를 생성하고 여러 속성을 설정하는 데 사용되어 코드를 깔끔하게 유지하는 데 이용되었다.

```
// let
inline fun <T, R> T.let(block: (T) -> R): R {
    return block(this)
}

// Usecase
val name: String? = null
name?.let {
    println("완벽한 이름")
} ?: println("null 입니당")
```

let 함수는 호출된 객체를 블록의 인자로 전달하고, 블록의 결과를 반환한다.

let은 null 가능성이 있는 객체에 대해 작업을 수행할 때 유용하다. ?. (세이프 콜) 연산자와 함께 사용될 수 있으며 null이 아닌 경우에만 코드 블록을 실행한다.

```
// run
inline fun <T, R> T.run(block: T.() -> R): R {
    return block()
}

// Usecase
val jiwon = shin.run {
    smart(100)
    lazy(50)
    increaseIQ()
}
```

run 함수는 호출된 객체의 context 에서 블록을 실행하고, 블록의 결과를 반환한다. 이는 코드 블록 내에서 여러 작업을 수행하고 마지막에 결과를 반환하고자 할 때 적합하다. 위 코드에서 run 은 smart, lazy 에 대한 특성 값을 수행하고 IQ 증가를 계산하는 일련의 연산을 수행하고 결과를 반환한다.

앞선 네 함수를 비교하는 표는 아래와 같다

호출시 수신 객체 입력 코드 블록으로 수신 객체 전달	Reciver 로 암시적 전달	
Reciver 로 암시적 전달	apply	run
Parameter 로 명시적 전달	also	let
반환	전달 받은 수신 객체	코드블록의 수행 결과

4.

A.

```
fun <T> List<T>.size(): Int =
    fold(0) { listSize, _ -> listSize + 1 }
```

제공된 fold(0) 은 초기값을 0으로 선언하여 List 를 fold 하겠음을 나타낸다. 각 요소를 지날 때마다 listSize 를 1씩 증가시켜서 List의 크기를 계산하는 코드이다. 요소의 값은 list 의 size 를 구하는 데 중요하지 않기 때문에 무시하고 listSize 만 증가시킨다. 이를 위해 _ 를 사용하여 요소 값을 무시한다.

B.

```
fun <T> List<T>.filter(p: (T) -> Boolean): List<T> =
    flatMap { if (p(it)) listOf(it) else emptyList() }
```

조건 p에 맞는 요소만을 포함하는 새로운 리스트를 생성하는 filter() 함수를 구현하기 위하여 위와 같이 코드를 작성하였다. { } 안에 코드는 주어진 조건 p 를 만족하는 경우에만 listOf 를 사용하여 해당 요소를 포함하는 리스트를 반환하고, 그렇지 않을 때는 비어있는 리스트를 반환하는 것을 의미한다. 최종적으로 flatMap() 함수는 p 를 만족하는 요소들로 이루어진 새로운 리스트를 반환한다.