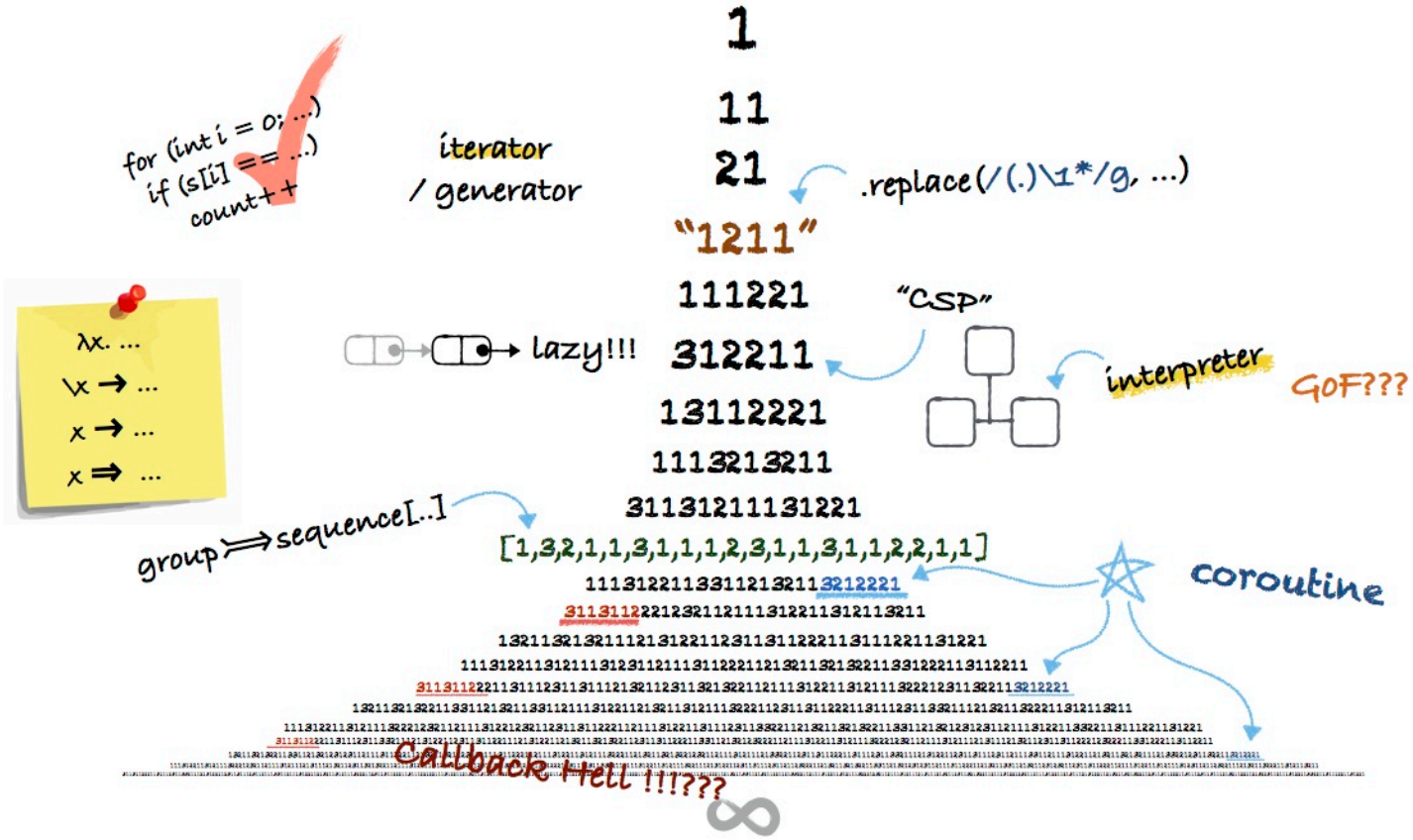
알아두면 피와 살이 될 프로그래밍 개념 10가지

개미 수열을 푸는 10가지 방법

한주영



개미 수열을 푸는 10가지 방법

알아두면 피와 살이 될 프로그래밍 개념 10가지

한주영

This book is for sale at http://leanpub.com/programming-look-and-say

This version was published on 2017-08-01

혜정이와 지민이에게

그리고,

개미 수열 문제에 참여해 준 많은 엘지전자 개발자들과 커뮤니티 멤버들에게

차례

1장 개미 수열 시작하기

9번째 줄에 나올 수는 무엇인가?

1

11

12

1121

122111

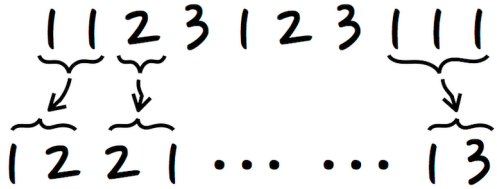
112213

12221131

1123123111

?

답은 122131112131113이다. 이 수열은 '개미 수열'[1]이라고 알려져 있으며, 앞 줄을 읽어서 다음 줄을 만들어낼 수 있는 무한 수열이다. 위 문제에서 마지막 줄을 "1이 2개, 2가 1개, 3이 1개, …" 처럼 읽을 수 있고, 따라서 "122131…" 의 결과를 얻게 된다.



개미 수열의 n번 줄 출력하기. 이 책이 풀고자 하는 문제다.

읽고 말하기 수열

개미 수열은 본래 '읽고 말하기 수열(look-and-say sequence)'[2]이라고 한다. 읽고 말하기 수열은 다음과 같이 전개된다.

---

[1] 1993년 베르나르 베르베르의 소설 '개미'의 번역서를 출판하면서 출판사가 이 소설에 등장하는 수열 퀴즈를 이벤트로 진행했다. 소설 개미가 국내에서 큰 인기를 얻으면서 이 수열은 유독 한국에서만 '개미 수열'로 불리고 있다.

[2] https://en.wikipedia.org/wiki/Look-and-say_sequence

1

11

21

1211

111221

수열이 만들어지는 규칙은 이미 설명한 대로다. 하지만 영어로 읽어야 한다. 다섯 번째 줄을 영어로 읽는 다면 “three 1s, two 2s, then one 1”이 되어 그 다음 줄 “312211”을 얻을 수 있다. 하지만 앞서 소개한 개미 수열은 이것과 조금 다르다.

1

11

12

1121

122111

우리는 “세 개의 1” 대신 “1이 세 개”처럼 읽으므로 수열의 형태가 바뀐 것이다. 재미있게도 이 수열은 읽는 방식을 뒤집는다 해도 전체적으로 영향을 받지 않고 단지 각 줄이 뒤집어질 뿐이다. 수열 본래의 특성이 그대로 유지된다.

우리나라에서는 읽고 말하기 수열보다는 각 줄이 뒤집어진 개미 수열이 더 유명하다. (우리에게 더 자연스러운 뒤집어진 수열은 분명 본래의 읽고 말하기 수열과는 다른 수열이다.) 이 두 수열을 구분지어 부를 수도 있겠지만, 이 책에서는 길이를 먼저 말하는 영어식 표현의 수열을 중심으로 다룰 것이다.

구글 랩스 적성 검사

개미 수열의 첫 다섯 줄을 보고 그 다음 줄을 답하는 문제가 2004년 구글 랩스 적성검사에 나왔다고 한다.[1] 이러한 유형의 문제를 통찰력 문제라고 하나 본데, 지금은 구글에서 이런 유형의 문제를 사용하지 않는 모양이다. 이런 문제에서 규칙을 찾아내는 것에는 무게를 두지 않겠다는 의미일 것이다. 하지만 규칙을 알아내는 것과 이 규칙을 어떻게 코드로 옮길 것인가 하는 것은 다른 문제다. 이 책은 개미 수열의 규칙을 다양한 방법을 동원하여 코드로 옮겨본다.

[1] **당신은 구글에서 일할 만큼 똑똑한가?**, 윌리엄 파운드스톤 지음, 유지연 옮김, 타임비즈, 2012

Java #0 - for와 String

개미 수열 문제를 푸는 첫 번째 프로그램을 살펴보자. 아래 Java 코드는 개미 수열의 10번 줄을 출력한다. (이제부터는 0번 줄을 첫 줄로 본다.)

개미 수열 10번 줄 출력 (Java)

```java
public class LookAndSay {
  public static void main(String... args) {
    String s = "1";
    for (int line = 0; line < 10; line++) {
      int length = 1;
      char head = s.charAt(0);
      String result = "";
      for (int i = 1; i < s.length(); i++) {
        if (s.charAt(i) == head) {
          length++;
        } else {
          result += length;
          result += head;
          length = 1;
          head = s.charAt(i);
        }
      }
      result += length;
      result += head;
      s = result;
    }
    System.out.println(s);
  }
}
```

첫 줄 "1"을 시작으로, 다음 줄의 계산을 반복한다. 다음 줄을 계산하는 방법은 앞 줄의 글자를 하나씩 읽으면서 이미 읽은 글자와 같으면 길이(length)를 증가시키고 그렇지 않으면 지금까지 읽은 글자의 길이와 그 글자(head)를 결과에 누적시킨다.

아름다운 코드가 아니어서 실망할지도 모르겠다. 하지만 개미 수열의 10번 줄을 출력하는 **완벽한** 프로그램이다. 이 코드의 목적은 이 책을 읽는 모두가 개미 수열 문제의 규칙을 **완전히** 이해할 수 있도록 돕기 위한 것이다. 이 코드는 앞으로 만들어나갈 다양한 변형의 중요한 베이스라인이 되어줄 것이다.

Haskell #0 - 리스트

단순한 문제에 단순한 코드를 보고 실망하신 분들을 위해 조금 난해한 코드를 하나 보여주겠다. 관심을 가지고 이 책을 계속 읽어나가는 데 조금이나마 도움이 되기를 바란다. 아래는 Haskell로 작성한 개미 수열 프로그램이다.[3] Java 프로그램과 마찬가지로 10번 줄을 출력한다.

개미 수열 10번 줄 출력 (Haskell)

```
import Data.List
import Control.Monad

ant = iterate (group >=> sequence[length, head]) [1]
main = print (ant !! 10)
```

이 프로그램은 ant, main 두 함수를 정의한다. ant 함수는 개미 수열을 리스트로 나타내고, main은 개미 수열의 10번 줄(!! 10)을 출력한다. ant 함수는 첫 줄 [1]을 시작으로 다음 줄 계산(group >=> sequence[length, head])을 반복(iterate)한다. 다음 줄을 계산할 때는, 먼저 앞 줄을 같은 값끼리 묶고(group), 각 그룹의 길이(length)와 머릿값(head)으로 바꿔치운다.(>=> sequence[])

Java 코드와 Haskell 코드는 본질적으로 같은 일(10번 줄 출력하기)을 하면서도 동작의 차이는 엄청나다. 이 책을 통해 Java나 JavaScript로도 그 차이를 극복할 수 있을 것이다.

Java #1 - ant()와 next() 분리

이미 보여준 Java 코드는 main()에 모든 내용이 들어있어서 뭔가 이야기를 이어가기가 어려우니 조금만 리팩터링을 해 놓자.

ant()와 next()로 분리 (Java)

```
1   private static String ant(int n) {
2     String s = "1";
3     for (int line = 0; line < n; line++) {
4       s = next(s);
5     }
6     return s;
7   }
8
9   private static String next(String s) {
10    int length = 1;
```

[3] https://www.haskell.org 에서 개발환경을 받아서 설치하면 예제 코드를 실행시켜볼 수 있다.

```
11      char head = s.charAt(0);
12      String result = "";
13      for (int i = 1; i < s.length(); i++) {
14        if (s.charAt(i) == head) {
15          length++;
16        } else {
17          result += length;
18          result += head;
19          length = 1;
20          head = s.charAt(i);
21        }
22      }
23      result += length;
24      result += head;
25      return result;
26    }
```

ant(n) 함수는 개미 수열의 n번 줄을 계산하는 함수고, next(s) 함수는 앞 줄을 읽어 다음 줄을 계산하는 함수다.

Wishful Thinking

일단 동작하는 main()을 만들고, 여기서 ant()를 Extract하고, 다시 여기서 next()를 Extract하였다. 또다른 접근법은 Wishful Thinking이란 것인데, "이런 것이 있다면…" 이란 가정으로 코드를 만들어가는 방법이다.[4] 잘 익히고 사용하면 매우 유용한 방법이다.

어떤 클래스가 있다면, 어떤 메쏘드가 있다면, 어떤 변수가 있다면… 이런 순간마다 필요로 하는 요소를 추가하기 위해 다른 위치로 점프하는 것은 사고의 흐름을 끊어버린다. 대신, 현재 그런 클래스나 메쏘드가 없다고 하더라도 흐름을 끊지 않고 마치 이미 다 준비되어 있는 것처럼 진행하는 방법이다.

마치 ant()가 이미 있는 것처럼 아래와 같이 작성한다.

```
System.out.println(ant(10));
```

ant(10)에 빨간불이 들어올 것이다. IDE의 자동 완성 기능은 멋지게 ant()함수의 기본 골격을 만들어 준다. 다음 줄을 계산하는 next()라는 함수가 있다면(물론 아직 만들지 않았다), ant()는 이를 n번 호출하면 된다.

[4] Structure and Interpretation of Computer Programs(아벨슨과 서스만)의 수많은 팁 중 하나다. https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-14.html#%_idx_1306

```
String ant(int n) {
  String s = "1";
  for (int i = 0; i < n; i++) {
    s = next(s);
  }
  return s;
}
```

다시 next(s)에 빨간불이 들어올 것이다. 자동 완성 기능으로 next()의 뼈대를 만든다. 처음부터 이런 방법으로 작업했다면 next()는 앞 절의 코드보다 훨씬 더 간결하고 읽기 쉬운 코드였을 것이다.

3장 리스트 처리

1장에서 우리는 개미 수열의 각 줄을 문자열로 보았다. 덕분에 2장에서 정규표현식을 이용하여 개미 수열의 다음 줄을 계산할 수 있었다. 하지만 더 일반적으로 보자면 각 줄을 **숫자 리스트**라고 보는 것이 타당할 것 같다. 말하자면 `next()`의 시그너처[5]를 다음처럼 보자는 얘기다.

```
List<Integer> next(List<Integer> ns)
```

이 장에서는 개미 수열의 다음 줄을 계산하는 `next()`의 동작을 하나하나 따져보고 **리스트 처리**라는 관점에서 다시 구현해 볼 것이다.

`next()`가 하는 일

2장의 정규표현식은 잊고 다시 1장의 끝에서 막 리팩터링한 `next()`가 하는 일을 하나씩 따져보자. `next()`는 여전히 여러가지 일을 한꺼번에 하고 있다.

- 루프 – 당연하게도 반복문을 포함하고 있다. 숫자를 하나씩 끝까지 읽어야 한다.
- 카운트 – 같은 숫자의 반복 횟수를 카운트한다.
- 비교 – 새로 읽은 숫자가 현재 카운트 중인 숫자와 같은지 비교한다.
- 문자열 생성 – 반복된 숫자의 갯수와 그 숫자를 이용하여 결과 문자열을 누적 생성한다.

각각의 일은 대수롭지 않지만 이런 일들을 한꺼번에, 그것도 서로 뒤엉킨 체로 처리한다는 점이 문제다. (애초에 `ant()`가 분리되기 전에는 전체를 감싸는 루프가 하나 더 있었다.)

Java #3 - 리스트 처리

`next()`를 리스트를 처리하는 함수로 본다면 다음과 같이 단계를 나누어 볼 수 있다. 예를 들어, [1,3,1,1,2,2,2,1]를 입력으로 가정해 보자.[6]

1. 리스트를 같은 숫자들의 그룹으로 나눈다. [1,3,1,1,2,2,2,1] → [[1],[3],[1,1],[2,2,2],[1]]
2. 각 그룹을 길이와 그룹의 숫자의 리스트로 바꾼다. → [[1,1],[1,3],[2,1],[3,2],[1,1]]
3. 리스트의 리스트를 다시 하나의 리스트로 이어붙인다. → [1,1,1,3,2,1,3,2,1,1]

---

[5] JavaScript는 함수의 타입을 명시하지 않지만, Flow나 TypeScript에서는 `function next(x: [number]): [number]`라고 볼 수 있다.

[6] 리스트 표기법은 JavaScript의 배열 표기법을 차용하였다. Haskell의 리스트 표기법이기도 하다.

각 단계를 도와주는 함수들이 이미 있다면, 다시 말해 이번에도 Wishful Thinking을 적용해 본다면 어떨까? 다음의 함수들이 있다면 next()를 구현하기 더 수월할 것이다.

List<List<A>> group(List<A> as)
: as의 요소들을 이어진 같은 요소들끼리 그룹을 만든다.[7]

List<B> map(Function<A,B> f, List<A> as)
: as의 요소들을 f를 이용하여 B로 만든다.

List<A> concat(List<List<A>> ass)
: ass의 요소들(리스트)을 이어붙여서 하나의 리스트를 만든다.

이런 함수들만 있다면 next()를 구현하는 것은 너무나 간단한 일이다. map()의 인자는 2장의 replace()처럼 함수를 인자로 받는다. 이 경우는 group()으로 나누어진 각 그룹에 대해 변환 함수를 인자로 넘겨주면 되는데, 그룹(리스트)의 길이와 그룹의 값으로 이루어진 리스트를 반환하는 함수여야 한다.

group/map/concat을 이용하는 next() (Java)

```
1  private static List<Integer> next(List<Integer> ns) {
2    return concat(map(g -> asList(g.size(), g.get(0)), group(ns)));
3  }
```

리스트 처리 함수들

우리가 있었으면 하고 바랐던 세 함수 중에서 map()과 concat()은 구현하기 어렵지 않다. 리스트의 각 요소들을 결과 리스트에 적절히 추가하기만 하면 된다.

리스트 처리 함수 – map과 concat (Java)

```
1  public static <A, B> List<B> map(Function<A, B> f, List<A> as) {
2    List<B> bs = new ArrayList<>();
3    for (A a : as) {
4      bs.add(f.apply(a));
5    }
6    return bs;
7  }
8
9  public static <A> List<A> concat(List<List<A>> ass) {
10   List<A> list = new ArrayList<>();
11   for (List<A> as : ass) {
```

[7] 혹은 group() 대신 List<RunLength<A>> runLength(List<A> as)를 기대할 수도 있을 것이다.

```
12        list.addAll(as);
13      }
14      return list;
15    }
```

그런데 group()은 중첩된 리스트를 생성해야 하므로 상대적으로 복잡하다.

리스트 처리 함수 – **group** (Java)

```
1   public static <A> List<List<A>> group(List<A> as) {
2     List<List<A>> ass = new ArrayList<>();
3     List<A> g = null;
4     for (A a : as) {
5       if (g == null || !g.get(0).equals(a)) {
6         g = new ArrayList<>();
7         ass.add(g);
8       }
9       g.add(a);
10    }
11    return ass;
12  }
```

group()의 구현을 보면 개미 수열의 next()에 포함될 로직과 비슷하다. 하지만 **일반화**되었다는 점이 다르다. 이제 group()은 그 자체로 다른 문제에 재사용될 수 있다.[8]

정규표현식의 replace()와 마찬가지로 이번 장에서 정의한 group(), map(), concat()은 개미 수열 문제에 국한되지 않으며 일반적인 리스트 처리에 사용할 수 있다. 각 함수의 시그너처에 타입 파라미터가 있다는 점만 보아도 알 수 있다.

App-specific	General
ant() next()	group() map() concat()

[8] group()은 더 기본적인 리스트 처리 함수들로 구현되는 것이 일반적이다. 9장 지연 리스트를 참고하라.

연습. JavaScript 배열 처리하기

JavaScript의 기본 자료구조인 배열에는 이미 `map()` 메쏘드가 정의되어 있다. `concat()` 메쏘드는 정의되어 있지만 우리가 기대하는 것과는 다르게 동작한다. 아래의 `next()`가 제대로 동작하도록, 중첩된 배열을 펼친 새로운 배열을 반환하는 `flatten()`이라는 메쏘드와 이어진 같은(`===` 비교) 요소끼리 묶어서 중첩 배열을 반환하는 `group()`이라는 메쏘드를 추가해 보라.

```
function next(ns) {
  return ns.group().map(g => [g.length, g[0]]).flatten()
}
```

함수형 프로그래밍

이 장에서 맛보기로 보여준 몇 개의 리스트 처리 함수들은 **함수형 프로그래밍**(Functional Programming)에서는 매우 일반화된 유틸리티 함수다. Java는 버전 8에 이르러서야 함수형 프로그래밍 요소들을 조금씩 도입하기 시작했지만, 현대적 언어라고 할 수 있는 거의 모든 언어들은 이미 이런 요소들을 포함하고 있으며, Java나 JavaScript에서 함수형 스타일을 사용할 수 있게 도와주는 라이브러리들도 많다.[9] 우리가 직접 고민하여 새로 작성할 필요가 없다. 이러한 라이브러리들이 기본적으로 제공하는 유틸리티의 추상 수준이 매우 높기 때문에 이번 장의 `next()`에는 오류가 개입될 여지가 매우 적다.

1장의 Haskell 코드에서 `ant` 함수의 정의를 다시 살펴보자.

```
ant = iterate (group >=> sequence [length, head]) [1]
```

여기서 `group >=> sequence [length, head]` 부분이 Java나 JavaScript로 작성했을 때 다음 줄을 계산하는 `next()`에 해당한다.[10]

이런! 벌써 Haskell과 거의 유사한 수준의 구현을 얻게 되었다. 사실 이번 장에서 리스트 처리 함수를 이용하여 작성된 `next()`는 함수형 프로그래밍 스타일을 따른 것이고, Haskell은 함수형 프로그래밍 언어의 대표 격이라서 두 코드가 유사할 수 밖에 없다. 그러나 결정적인 차이는 아직 남아있으니 여기서 멈추지 말기를 바란다.

---

[9] TotallyLazy, Ramda.js 등이 있다.

[10] `next = group >=> sequence [length, head]`로 정의하고, `ant = iterate next [1]`로 바꿀 수 있다. `next` 함수는 `concat . map (\g -> [length g, head g]) . group`로 정의할 수도 있다.

4장 이터레이터

이 장에서는 개미 수열을 이터레이터로 접근해 본다. 왜 갑자기 이터레이터를 이야기하는지 배경부터 살펴보자.

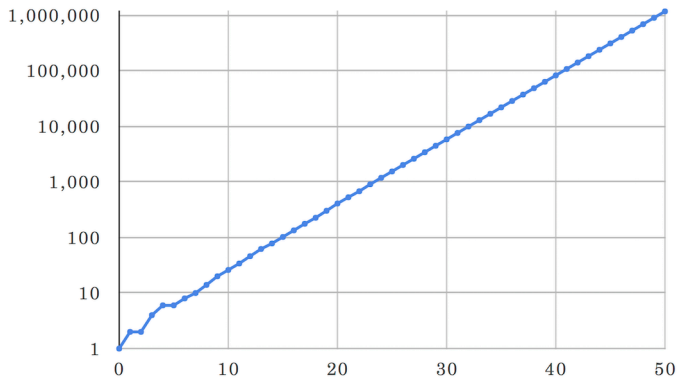
개미 수열의 특성

먼저 개미 수열의 100번 줄을 출력해 보자. 지금까지 만들어본 Java 프로그램으로 100번 줄을 출력하고자 하면 한참 시간이 지나 아래의 결과를 보게 된다.

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

JavaScript라고 다를 것은 없다. 고작 100번 줄을 출력하는데 메모리가 모자란다. 반면 1장에서 보여준 Haskell 프로그램은 100번 줄을 문제없이 출력한다. 3장에서 개미 수열을 리스트 처리 관점으로 구현하면서 Haskell과 거의 동등한 모양을 갖추긴 했지만 동작에 있어서는 큰 차이가 있음을 알 수 있다.

개미 수열은 각 줄이 약 30%씩 길어지는 특성이 있다.[11] 따라서 100번 줄의 길이는 666,450,031,706이며, 숫자 하나를 1바이트로 보아도 100번 줄만을 위해 약 600G 이상의 메모리가 필요하다. 당연히 메모리가 모자랄 수 밖에 없다.[12]



개미 수열의 길이 증가 (로그 스케일)

개미 수열 한 줄 전체를 문자열이나 리스트에 담을 수 없다면 **리스트를 나타내는 새로운 방법**을 써야 한다. 대표적인 방법이 이터레이터다.

---

[11] 증가 비율의 극한값은 $\lambda = 1.303577269034\cdots$이다.

[12] 계산하는데 개인 노트북에서 24시간이 걸렸다.

이터레이터와 이터러블

이터레이터(iterator)는 이미 오래전부터 많은 언어들이 기본으로 제공하는 기능이다. 이터레이터는 리스트나 트리처럼 값을 저장하고 있지는 않지만, 이들 자료 구조의 값을 순차적으로 참조할 수 있는 일관된 방법을 제공한다. 이터레이터를 제공하는 컬렉션은 이터러블(iterable)이라 한다.

JavaScript의 이터러블/이터레이터 프로토콜
: 이터러블 객체는 `Symbol.iterator` 키에 이터레이터를 반환하는 함수를 가진다. JavaScript 이터레이터는 `next()` 메쏘드를 가지는데, 호출할 때마다 값을 반환하거나 종료 여부를 알린다.

Java의 `Iterable`/`Iterator` 인터페이스
: `Iterable` 인터페이스는 `iterator()` 메쏘드가 있어서, 이를 통해 이터레이터를 생성한다. `Iterator` 인터페이스는 `hasNext()`로 종료 여부를 판단하고, `next()`로 값을 얻어온다.

JavaScript 이터레이터의 `next()` 메쏘드는 Java의 `Iterator` 인터페이스에 있는 두 개의 메쏘드를 합쳐놓은 것이다. JavaScript와 Java는 각각 `for` 문에서 이터러블을 순회할 수 있도록 문법적으로 지원한다.

일반적으로 이터러블을 컬렉션, 그리고 이터레이터는 컬렉션을 순회하기 위한 인터페이스라고 봐도 되지만, 항상 그렇지만은 않다. 이터레이터는 무언가를 순차적으로 하나씩 살펴보기 위한 범용적인 인터페이스일 뿐이다. 예를 들어, Java에서 입력 스트림의 텍스트 처리를 위한 `Scanner` 클래스는 `Iterator`를 구현하고 있다. `Scanner`를 이용하기 위해서 사용자의 입력을 모두 기다리거나 메모리에 올려둘 필요가 없다.

개미 수열과 이터레이터

개미 수열 문제에서 핵심이 되는 것은 다음 줄을 계산하는 `next()` 함수가 구현하는 로직이다. 1장의 `for` 루프 구현을 보면 앞 줄에서 한 글자씩 읽으면서 다음 줄을 만들어 낸다.

개미 수열 49번 줄의 길이는 약 90만이고 다음 줄인 50번 줄의 길이는 110만을 넘는다. 50번 줄의 첫 숫자를 계산하려면 49번 줄에서 몇 글자만 알면 될까? 단지 첫 네 숫자만 알면 된다. 49번 줄은 "1113…" 으로 시작하는데, 3을 읽으면 50번 줄이 "31…" 으로 시작하는 것을 알 수 있다.

```
49: 1113122113121...
50: 3113112221131...
```

마찬가지로 50번 줄의 100번째 숫자를 계산하는 경우를 보자. 49번 줄의 70번째 위치의 "…32112…" 부분에서 가운데 "11"로 결정되는 "21"이 50번 줄의 100번째 숫자가 된다. 즉, 75번째 숫자까지만 읽으면 벌써 100번째 숫자까지 계산이 된다. 비슷하게 49번 줄의 75번째 숫자 2를 계산하려면 48번 줄에서는 53개의 숫자까지 계산하면 된다.

```
48: ...331121...                          (마지막 1는 53번째 숫자)
49:         ...232112...                  (마지막 2는 75번째 숫자)
50:                   ...12131221...      (마지막 1은 100번째 숫자)
```

이렇게 50번 줄을 계산하기 위해 길이가 90만이 넘는 49번 줄 전체를 참조하지 않는다. 다만 순차적으로 하나씩만 참조할 수 있으면 된다. 그리고 이미 읽고 처리한 값들을 따로 저장해 둘 필요도 없다.

개미 수열은 이터레이터를 이용하기 딱 좋은 경우다. 입력으로 한 줄 전체가 아니라 이터레이터를 받아도 되며, 물론 다음 줄도 이터레이터로 전달하면 된다. 50번째 줄의 이터레이터를 순회할 때, 그 이전 줄들의 이터레이터도 함께 진행되며, 각 이터레이터는 마치 커서처럼 해당 줄의 데이터를 다음 이터레이터로 하나씩 전달한다.

JavaScript #1 - 이터레이터

이제 JavaScript 이터레이터를 이용하여 개미 수열을 구현해 보자.

개미 수열의 각 줄을 이터레이터로 나타내면, 개미 수열의 다음 줄을 계산하는 `next()` 함수는 이터레이터(앞 줄)를 입력으로 받아서 새로운 이터레이터(다음 줄)를 만들어야 한다. `next()`를 곧바로 구현할 수도 있겠지만[13] 3장의 추상화를 재사용할 수도 있다.

이터레이터를 이용한 `ant()`와 `next()` (JavaScript)

```
function ant(n) {
  let s = iter([1])
  for (let i = 0; i < n; i++) {
    s = next(s)
  }
  return s
}

function next(ns) {
  return concat(map(g => iter([g.length, g[0]]), group(ns)))
}
```

사실 앞의 코드는 3장의 리스트 처리 버전과 거의 똑같다. 어차피 문제의 본질은 바뀌지 않았고 데이터 표현만 리스트나 배열이 아닌 이터레이터로 바뀐 것이다.

이제 이터레이터에 대해 동작하는 `group()`, `map()`, `concat()`을 만들면 된다. 세 함수는 모두 이터레이터를 입력으로 받아서 새로운 이터레이터를 만들어 반환한다. JavaScript에서 이터레이터를 생성하는 함수는 아래와 같은 기본 뼈대를 가진다.[14]

[13] 이터레이터 버전의 `next()`를 한 덩어리로 구현하는 것도 가능하다. 직접 구현하고 비교해 보자.

[14] 이터레이터는 성능 상의 이유로 클래스로 만드는 것이 더 일반적이다.

이터레이터 생성 함수 뼈대 (JavaScript)

```
function makeIterator() {
  return {                // 이터레이터 객체
    next() {
      return {done: true} // 혹은 {done: false, value: ...}
    }
  }
}
```

리스트 처리에서 구현했던 것보다 조금 더 어려울 것이다. 가장 쉬운 map()부터 하나씩 살펴보자.

이터레이터 map()

map()은 변환 함수와 이터레이터를 인자로 받아서 새로운 이터레이터를 반환한다. 입력 이터레이터가 종료될 때 출력 이터레이터도 종료되고 값도 일대일 매칭되기 때문에 어렵지 않다.

이터레이터 map() 함수 (JavaScript)

```
function map(f, it) {
  return {
    next() {
      let {value, done} = it.next()
      if (done) {
        return { done: true }
      } else {
        return { done: false, value: f(value) }
      }
    }
  }
}
```

이터레이터 concat()

concat()은 중첩된 이터레이터(이터레이터의 요소가 다시 이터레이터)를 순환하므로 종료가 조금 까다롭다.

이터레이터 `concat()` 함수 (JavaScript)

```
function concat(it) {
  let inner = null
  return {
    next() {
      while (true) {
        if (inner === null) {    // 다음 내부 이터레이터 찾기
          let {value, done} = it.next()
          if (done) {
            return { done: true } // 외부 이터레이터가 끝나면 종료
          } else {
            inner = value
          }
        }
        let {value, done} = inner.next()
        if (done) {
          inner = null
        } else {
          return { done: false, value } // 내부 이터레이터의 다음 값 반환
        }
      }
    }
  }
}
```

입력 이터레이터 `it`에서 꺼낸 내부 이터레이터를 순환하여야 하므로 상태로서 저장해둬야 한다. 내부 이터레이터 `inner`가 종료된 경우에는 곧바로 다음 이터레이터를 꺼낼 수 있어야 하므로 루프도 필요하다.

이터레이터 `group()`

`group()`이 반환하는 이터레이터는 `next()`가 호출될 때마다 입력 이터레이터를 1~n번 진행한다. 그런데 동일한 요소들을 묶은 그룹을 하나 반환하려면 반환하는 그룹에 포함되지 않는 요소를 하나 더 읽은 상태가 되며, 이 상태 때문에 구현이 까다롭다.

이터레이터 `group()` 함수 (JavaScript)

```
function group(it) {
  let g = null
  return {
    next() {
      while (true) {
        let {value, done} = it.next()
        if (done && g === null) {
          return { done: true }
        } else if (done) {
          let result = g
          g = null
          return { done: false, value: result }
        } else if (g === null) {
          g = [value]
        } else if (g[0] === value) {
          g.push(value)
        } else {
          let result = g
          g = [value]
          return { done: false, value: result }
        }
      }
    }
  }
}
```

입력 이터레이터가 종료된 경우라도 이미 읽은 값이 있는 경우와 그렇지 않은 경우(처음부터 빈 이터레이터)를 고려해야 한다.

이터레이터 도움 함수 `iter()`와 `uniter()`

대개는 직접 이터레이터를 사용하기 보다는 컬렉션(이터러블)을 사용한다. 우리는 이터레이터를 바로 사용하므로 이터러블을 요구하는 다른 코드들(예를 들어 `for`)과 매끄럽게 연결되지 않는 문제가 있다. 이런 문제는 `iter()`와 `uniter()`같은 도움 함수로 해결할 수 있다. `iter()`는 이터러블을 이터레이터로 바꿔주고, `uniter()`는 이터레이터로부터 이터러블을 만들어준다.

이터레이터와 이터러블을 상호 변환하는 도움 함수 (JavaScript)

```
function iter(obj) {
  return obj[Symbol.iterator]()
}

function uniter(it) {
  return {
    [Symbol.iterator]: function () {
      return it
    }
  }
}
```

이제 아래의 코드로 개미 수열의 100번 줄을 출력해 볼 수 있다.

이터레이터 개미 수열 출력하기 (JavaScript)

```
for (let a of uniter(ant(100))) {
  process.stdout.write( `${a}` )
}
```

메모리 문제없이 잘 출력되는지 직접 확인해 보자. :-)

```
$ node look-and-say.js
11131221131211132221232112111312111213111213211231132132211211131221232112111312
21121311121312211213211321322112311311222113311213212322211211131221131211221321
12311321322112111312211312111322211213111213122112132113121113222112132l...
```

지연 리스트로서의 이터레이터

한 번에 모두 담을 수 없는, 사실상 무한의 길이를 가진 리스트를 이터레이터로 표현하였다. 이러한 리스트를 **지연 리스트**라고 한다. 이터레이터는 지연 리스트 혹은 무한 리스트의 일종이다. 이터레이터의 `next()` 메쏘드를 호출할 때까지 리스트의 꼬리 부분 계산은 지연된다.

일반적인 리스트와 구분하기 위하여 지연 리스트를 **스트림**(stream)이라고 부르는 경우도 있다. Java 8의 `Stream` 역시 무한대를 표현할 수 있는데, 그 밑에는 바로 이터레이터가 깔려있다.

연습. Java 이터레이터 구현하기

개미 수열의 각 줄을 `Iterator<Integer>`라고 보면, 다음 줄을 계산하는 `next()`의 타입은 아래와 같다.

```
Iterator<Integer> next(Iterator<Integer> ns)
```

`next()`를 구현해 보라. 잘 동작하는지 확인하기 위해 개미 수열의 100번 줄을 출력해 보자.

이터레이터 구현 리뷰

이터레이터를 지연 리스트로 사용함으로써 개미 수열의 100번 줄을 출력할 수 있었다. 하지만 구현이 복잡하다. 게다가 이미 확인해 본 사람도 있겠지만, n을 1,000 혹은 10,000 이상으로 늘려보면 더 이상 동작하지 않는다.

먼저 이터레이터 구현이 복잡한 이유부터 살펴보자. 지연 리스트의 다음 값을 하나 계산하여 반환하기만 하면되는데 왜 복잡한 것일까? 이유는 이터레이터가 루프의 제어권을 가지고 있지 않다는 점이다. 이터레이터는 값을 생성하는 로직과 루프를 분리하였다. 값을 생성하는 로직이 단순하여 루프를 반복할 때마다 값을 하나씩만 생성하는 경우(예, `map()`)는 이터레이터를 만드는 것도 간단하다. 해당 루프의 몸체만 추출하면 된다. 매우 자연스럽다. 하지만 요소 값을 생성(혹은 접근)하는 로직이 선형적이지 않은 경우에는 이터레이터가 자체적으로 상태 정보를 관리해야 한다. 예를 들어 이 장에서 구현한 `concat()`이나 `group()`을 보면, 각각 상태 정보를 가진다. `next()` 호출 시 다음 값을 계산하려면 마지막 상태를 따지거나 추가적인 루프가 필요하다.[15] 트리나 그래프 자료 구조에 대한 이터레이터 구현이 복잡한 이유도 마찬가지다. 애초에 이터레이터 인터페이스가 나오게 된 이유가 이러한 복잡성을 감추고 단순한 인터페이스를 노출하기 위한 것이다.

이터레이터 구현이 복잡해지는 문제는 빈번하게 발생하기 때문에 언어 차원에서 좀더 쉬운 방법을 제공하기도 한다. 제너레이터(generator)가 그것이다. 5장에서 제너레이터를 살펴볼 것이다.

n이 커질 때 드러나는 한계는 조금 더 근원적인 것이다. 이 문제를 해결하기 위한 방법은 6장 코루틴부터 다룬다.

[15] 마이클 잭슨의 "구조 불일치(Structure Clash)"에 해당한다.

6장 코루틴

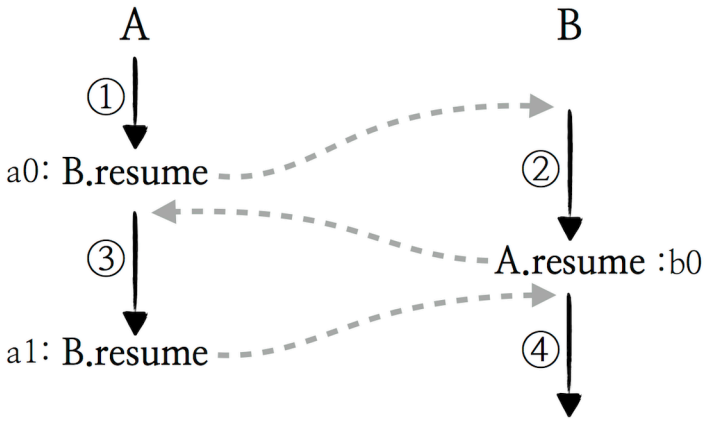
이 장에서는 제너레이터를 설명하면서 갑자기 등장한 코루틴이 무엇인지 살펴보고, 개미 수열에 응용해 볼 것이다.

코루틴 이해하기

위키백과사전의 제너레이터 설명에서 우리는 **코루틴**(coroutine)이라는 새로운 개념을 만났다. 링크를 따라가보면 위키백과사전 코루틴[16] 항목의 첫 문단은 다음과 같다.

> 코루틴은 비선점 멀티태스킹을 위해 서브루틴을 일반화한 컴퓨터 프로그램의 구성요소다. 실행을 중단하고 재시작할 수 있는 진입점을 여러 개 가질 수 있다. 코루틴은 협력적 멀티태스킹, 예외, 이벤트 루프, 이터레이터, 무한 리스트, 파이프와 같은 좀더 친숙한 프로그램 구성요소들을 구현하기에 적합하다.
>
> – 위키백과사전 "코루틴"

다른 함수를 호출하면 해당 함수로 점프하면서 콜스택이 쌓이고, 호출된 함수가 종료되면 원래 호출했던 위치로 되돌아 온다는 것은 너무도 당연한 이야기다. 그런데 코루틴의 경우엔 다르다. A 코루틴이 a0 지점에서 B 코루틴을 호출했다가, B 코루틴의 b0 지점에서 A를 **호출**(**혹은** resume)하면 a0 지점에서 실행이 재개되고, 다시 a1 지점에서 B 코루틴을 재개하면 b0 지점에서 실행을 이어간다. 어느 순간부터는 A가 상위 루틴인지 B가 상위 루틴인지 알 수 없다.



A와 B 코루틴의 실행

[16] https://en.wikipedia.org/wiki/Coroutine

제너레이터는 코루틴과 달리 실행을 중단할 때 호출한 함수로만 되돌아간다. 하지만 일반적인 함수와는 달리 중단된 지점에서 실행을 재개할 수 있다. 이러한 의미에서 제너레이터는 **반쪽짜리 코루틴**이라고 볼 수 있는 것이다.

Go의 고루틴과 채널

그런데, 두 개의 함수가 함께 실행되면서 제어권을 주고 받는다는 코루틴의 설명을 따르는 프로그래밍 모델을 어디선가 본 적이 있다. 게다가 이름도 매우 비슷하다. 바로 Go의 고루틴(goroutine)이다.

Go에는 고루틴이라는 특별한 순차 처리 개념이 있다. 고루틴은 마치 쓰레드처럼 동작하며, 여러 개의 고루틴은 자체 스케줄러를 통해 더 적은 갯수의 쓰레드에서 실행된다. 고루틴은 채널을 매개로 하여 동기화하거나 데이터를 교환한다.[17]

채널로 값을 보내려는 고루틴은 다른 고루틴이 같은 채널에서 값을 가져갈 때까지 블록되며, 보내고 받는 것은 동기적으로 이뤄진다. 반대로 먼저 받으려 시도할 때에도 다른 고루틴이 값을 보낼 때까지 블록되었다가 동기적으로 보내기/받기가 이뤄지면서 실행을 이어간다.

즉, 두 개의 고루틴은 하나의 채널을 공유하여 데이터를 보내고 받는 것으로 마치 코루틴처럼 제어권을 주고 받을 수 있다.

채널과 고루틴의 Hello World (Go)

```
1  func main() {
2          c := make(chan int)
3          go func() {
4                  <-c
5                  print( ” World ” )
6          }()
7          go func() {
8                  print( ” Hello ” )
9                  c <- 0
10         }()
11         time.Sleep(1000)
12 }
```

go 키워드로 시작되는 함수 호출은 새로운 고루틴을 만들어서 해당 함수를 실행한다. 위 코드에서 main() 함수는 “ World”를 출력하려는 고루틴과 “Hello”를 출력하려는 고루틴을 생성한다. “ World”를 출력하려는 고루틴이 먼저 시작된다 하더라도 채널에서 값을 읽을 때(<-c) 블록되어 더이상 진행하지 않는다. 이 블록이 해제되는 것은 두 번째 고루틴이 “Hello”를 출력한 뒤에 채널로 값을 하나 보낼 때(c <- 0)이다. 이로 인해 두 고루틴은 “Hello World”를 정상적으로 출력하게 된다. 동시에 실행되는 두 개의 고루틴이 채널 연산을 통해 협력적으로 동기화하는 모양새는 코루틴의 실행 모습과 매우 비슷하다.

[17] Go에서는 이러한 방식을 “Don’t communicate by sharing memory, share memory by communicating.”라는 말로 선전한다.

Go의 제너레이터 패턴

Go는 이터레이터 혹은 제너레이터라는 개념을 따로 정의하지 않는다. 하지만 기본적으로 제공되는 채널과 고루틴이라는 동시성 기능을 이용하면 쉽게 제너레이터를 만들 수 있다. 이미 5장에서 본 Java 제너레이터는 쓰레드를 이용하여 구현된 것이었다. 쓰레드라는 자원을 사용하기 때문에 스택이 넘치는 문제는 없지만 대신 메모리 부족이라는 문제가 있었다. 다행히 고루틴은 쓰레드처럼 생각할 수 있지만 그보다 훨씬 값싼 요소이다. Go의 고루틴 소개를 보면 수백만 개를 만드는 것도 가능하다고 한다. 즉, 제너레이터 스타일을 Go의 채널과 고루틴으로 구현한다면 개미 수열의 10,000번 줄, 혹은 수백만 번째 줄도 출력할 수 있을 것 같다.[18]

Go의 채널은 동기화 메커니즘이면서 실제로 데이터를 주고 받을 수 있는 매개이다. 값을 보내거나 받을 수 있는데, 값을 받는 쪽에서 보자면 이터레이터처럼 사용할 수 있고, 값을 보내는 쪽에서 보자면 제너레이터처럼 사용할 수 있다. 롭 파이크의 "Go 동시성 패턴" 발표[19]에서도 제너레이터를 첫 번째로 언급하고 있다.

boring() 함수는 제너레이터 (Go)

```
func main() {
    c := boring("boring!")              // 채널을 반환하는 함수
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c)
    }
    fmt.Println("You're boring; I'm leaving.")
}

func boring(msg string) <-chan string { // 읽기 전용 채널을 반환
    c := make(chan string)
    go func() {    // 함수 내부에서 고루틴을 새로 시작
        for i := 0; ; i++ {
            c <- fmt.Sprintf("%s %d", msg, i)
            time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
        }
    }()
    return c
}
```

boring() 함수는 문자열을 읽을 수 있는 채널(<-chan string)을 반환한다. 이는 마치 JavaScript의 제너레이터 함수를 호출하여 제너레이터 객체를 반환하는 것과 비슷하다. main()에서는 루프를 돌면서 채널에서 값을 읽어서(<-c) 출력한다. 이 채널로 값을 전달하는 것은 boring() 함수가 내부에서 시작한 고루틴이다. main()을 실행 중인 고루틴과 boring() 내부에서 분기한 고루틴은 c 채널을 공유하며 데이터를 주고 받는다.

[18] Go 개발팀은 쓰레드와 달리 고루틴을 수백만 개 만들 수 있다고 자랑한다.

[19] https://talks.golang.org/2012/concurrency.slide#25

이렇게 읽기 채널을 반환하면서, 내부에서 따로 고루틴을 만들어 그 채널로 값을 보내는 함수 형태가 제너레이터 패턴이다.

Go #0 - 채널과 고루틴

개미 수열을 Go의 고루틴과 채널로 구현하자면 각 줄을 채널로 나타내고, 앞 줄을 읽어 다음 줄을 생성하는, 제너레이터 패턴의 함수를 만들어서 n번 반복하면 된다. 즉, `next()` 함수는 n-1번 줄을 나타내는 채널을 입력으로 받아서 n번 줄을 나타내는 채널을 반환하고, 내부에서는 n-1번 채널의 값을 읽어서 n번 채널에 값을 쓰는 고루틴을 새로 시작해야 한다.[20] n번째 줄을 얻으려면 n개의 고루틴이 있어야 한다. 개미 수열의 첫 줄에 해당하는 채널에는 1 하나만 쓰여진다. 이터레이터/제너레이터는 데이터가 더 이상 없음을 알릴 수 있는데, Go에서는 채널을 닫는 것으로 끝을 알릴 수 있다.

먼저 n번 줄을 만들기 위한 `ant()` 함수는 첫 줄을 생성한 다음 채널을 `next()`로 n번 감싸준다. 다르게 표현하자면 여기서 이미 n개의 고루틴을 시작하고 고루틴들을 모두 채널로 연결시키는 셈이다.

n번 줄을 만들어주는 `ant()` (Go)

```go
func ant(n int) <-chan int {
        c := gen(1)
        for i := 0; i < n; i++ {
                c = next(c)
        }
        return c
}
```

첫 줄은 1을 보내고 채널을 닫으면 된다. Go의 채널 연산은 기본적으로 동기적이기 때문에 다음 줄을 계산하는 고루틴이 값을 읽기 전에는 보내기 연산이 블록된다.

첫 줄을 출력하는 `gen()` (Go)

```go
func gen(n int) <-chan int {
        c := make(chan int)
        go func() {
                c <- n
                close(c)
        }()
        return c
}
```

다음 줄을 출력하는 `next()` 함수는 기존의 `for` 구현이 그대로 사용되고 있다. 채널을 생성하고 고루틴을 실행시키는 기본적인 제너레이터 패턴을 따르고 있다.

[20] 읽기 채널을 받아서 새로운 읽기 채널을 반환하므로 채널에 대한 Transducer라고 할 수 있다.

다음 줄을 계산하는 **next()** (Go)

```
func next(in <-chan int) <-chan int {
        out := make(chan int)
        go func() {
                prev, count := <-in, 1
                for value := range in {
                        if value == prev {
                                count++
                        } else {
                                out <- count
                                out <- prev
                                prev, count = value, 1
                        }
                }
                out <- count
                out <- prev
                close(out)
        }()
        return out
}
```

이 Go 코드는 동시성 관점에서 보자면 5장에서 살펴본 쓰레드 제너레이터를 이용하는 Java 버전과 똑같다. 다만 고루틴이 쓰레드에 비해 생성 비용이 매우 낮기 때문에 이 Go 코드는 10,000번 줄, 심지어 1,000,000번 줄도 출력할 수 있다.

제너레이터 방식을 따르고 있기 때문에 구현은 단순해지면서, 고루틴들이 채널을 통해 모두 엮이면서, 마지막 채널(n번 줄)에서 값을 읽어가는 만큼만 계산이 이뤄진다. 지연 리스트의 효과도 그대로 얻은 것이다.

고루틴과 코루틴

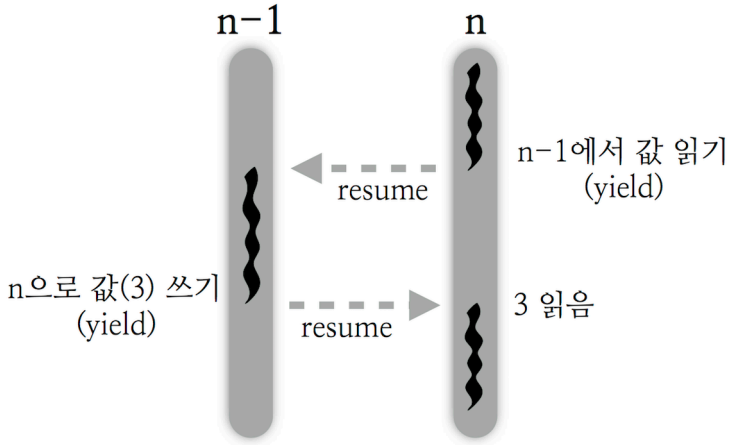
고루틴은 사실 코루틴이 아니다. 하지만 Go에서 권장하는 고루틴들 간의 동기화 메커니즘이 동기적 채널 통신을 이용하고 있다는 점은 코루틴의 동작을 연상케 한다. 고루틴 두 개가 채널로 동기화되며 실행되는 모양을 보면 코루틴 두 개가 협력적으로 실행되는 모양새와 같다. 코루틴은 쓰레드를 이용하지 않으면서 **협력적 멀티태스킹**을 구현하는데 사용되며, yield/resume으로 제어 흐름을 주고 받는다. 고루틴이 채널을 통해 동기적으로 값을 주고 받는 과정은 yield/resume으로 볼 수 있다. 또, 여러 개의 고루틴이 더 적은 갯수의 쓰레드에 멀티플렉싱되어 실행된다는 점, 그래서 쓰레드 하나에서 동작할 수 있다는 점 또한 코루틴과 유사한 점이다.

5장에서 콜스택을 사용하는 제너레이터와 쓰레드를 사용하는 제너레이터는 개미 수열의 10,000번 줄을 출력하기에 역부족이었다. 고루틴 혹은 코루틴을 흉내낸다면 JavaScript나 Java에서도 10,000번 줄을 출력할 수 있을 것이다.

고루틴을 흉내낸다는 것은 단일 쓰레드에 멀티플렉싱되는 순차 프로세스를 표현하고 이들을 채널을 이용하여 동기화하는 것이다. 즉 간단한 스케줄러를 구현해야 한다는 얘기다. 코루틴을 흉내내는 것도 거의 비슷하다. 멈췄다가 재시작 가능한 프로세스를 구현하고, 프로세스를 실행시켜 줄 디스패쳐를 만들면 된다. 디스패쳐는 한 번에 하나의 프로세스를 실행하면서 하나가 멈추면 적절히 다음 프로세스를 선택하여 실행한다.

코루틴과 개미 수열

5장에서 JavaScript 제너레이터를 이용한 것이나 이 장에서 Go로 구현한 개미 수열의 동작은 거의 비슷하다. 코루틴 관점에서 실행 흐름을 다시 정리해 보자. 각 줄을 생성하는 함수가 코루틴으로 동작하고, n번 줄과 n-1번 줄은 입출력이 서로 연동된다. 파이프처럼 동작하는 코루틴들은 선후 관계가 있어서 다음 실행할 코루틴은 쉽게 결정된다.



읽기/쓰기로 연동된 코루틴

1. n번 코루틴은 입력을 읽고자 할 때 yield한다.
2. 디스패쳐는 n-1번 코루틴을 resume한다.
3. n-1번 코루틴은 출력을 만들면서 yield한다.
4. 디스패쳐는 n번 코루틴을 resume한다. (3번의 출력을 읽어간다.)

C #0 - 코루틴

위에서 설명한 과정을 C로 어떻게 구현할 수 있을까? yield하면서 실행 위치를 저장하고, resume할 때 저장해 둔 실행 위치를 넘겨 받아 점프하면 된다. 또, 개미 수열의 `next()` 함수에서 사용하던 지역 변수들은 코루틴이 재개할 때 유지되어야 하므로 힙에 상태를 따로 저장해 두어야 한다. 아래와 같은 구조체를 얻을 수 있다.

코루틴 상태를 저장하기 위한 구조체 (C)

```c
typedef struct proc proc;
struct proc {
  int (*proc)(proc*); // 코루틴 함수 포인터
  char ptr;           // 재진입 위치
  char next;          // read 로 읽은 값
  char prev;          // 이미 읽은 값 (갯수를 세는 값)
  char count;         // 반복된 글자 갯수
};
```

proc과 ptr은 모든 코루틴이 가져야 하는 속성이며, next, prev, count는 next() 함수의 상태를 저장하기 위한 것이다.

개미 수열의 첫 줄 1을 출력하는 코루틴 함수는 간단하다. 사실 아래 함수는 상태를 저장할 필요가 없으므로 ptr 만 있어도 되지만 편의상 next()와 같은 구조체를 사용했다.

1을 출력하고 종료하는 코루틴 (C)

```c
int init(proc *s) {
  switch (s->ptr) {
    case 0:
      s->ptr++;
      return 1;
    default:
      return 0;
  }
}
```

처음 호출될 때 ptr은 0이며, 1을 출력하고(return 1), 다음 진입 시(ptr이 1)에 코루틴을 종료한다(return 0). 코루틴을 제대로 구현하려면 yield하는 이유(read 혹은 write)와 출력하는 값을 따로 처리해야 한다. 하지만 개미 수열에서 모든 출력값(혹은 입력값)이 1, 2, 3 중 하나라는 사실을 이용하여 약간의 트릭을 사용했다. 값을 읽기 위해 yield하는 경우는 -1을 반환하고(선행 코루틴을 재개한다) 코루틴을 종료하는 경우에는 0을 반환하게 했다(후행 코루틴을 재개한다). 그 외의 값은 yield하면서 출력하는 값이다(후행 코루틴을 재개한다).

다음 줄을 계산하는 next() 코루틴은 값을 읽기도 하고 출력하기도 하므로 조금 복잡하다. return -1을 next = read()라고 읽으면 조금 따라가기 쉽다. 코루틴을 떠나기 전에 재진입 위치를 ptr에 저장하므로 ptr = 3을 goto 3으로 볼 수 있다.

개미 수열의 앞 줄을 읽어 다음 줄을 출력하는 코루틴 (C)

```c
int next(proc *s) {
  int prev;
  switch (s->ptr) {
    case 0:
      s->ptr = 1;
      return -1;
    case 1:
      s->prev = s->next;
      s->count = 1;
      s->ptr = 2;
      return -1;
    case 2:                              // 루프 시작점 역할을 한다
      if (s->next == 0) {
        s->ptr = 5;    // 루프 탈출
        return s->count;
      } else if (s->prev == s->next) {
        s->count++;
        return -1;
      } else {
        s->ptr = 3;
        return s->count;
      }
    case 3:
      prev = s->prev;
      s->prev = s->next;
      s->count = 1;
      s->ptr = 4;
      return prev;
    case 4:
      s->ptr = 2;
      return -1;                         // 루프 반복
    case 5:
      s->ptr = 6;
      return s->prev;
    case 6:
      return 0;                          // 코루틴 종료
    default:
      assert("Unreachable");
      return 0;
  }
}
```

이제 `init()`과 `next()`를 이용하여 코루틴을 실행해 보자.

코루틴들을 실행시키는 `main()` (C)

```c
// 코루틴 준비
proc* procs = (proc*)calloc(n + 1, sizeof(proc));
procs[0].proc = &init;
for (int i = 1; i < n + 1; i++) {
  procs[i].proc = &next;
}

// 디스패치 루프 시작
int cur = n;
while (cur < n + 1) {
  int result = procs[cur].proc(&procs[cur]);
  if (result == -1) {
    cur--;
  } else if (cur < n) {
    cur++;
    procs[cur].next = result;
  } else if (result != 0) {
    printf("%d", result);
  } else {
    printf("\n");
    break;
  }
}
```

개미 수열의 코루틴은 의존성이 선형적이기 때문에 배열에 저장하고 디스패치 루프에서 인덱스를 증가/감소하는 것만으로 쉽게 코루틴을 선택하여 실행을 재개할 수 있다. 제대로 구현하고자 한다면 코루틴이 yield하면서 실행시킬 코루틴을 지정해 주도록 하면 된다. Go의 고루틴 스케줄링 방식을 따르자면, 고루틴들 간의 의존성을 채널로 구체화시키고, 고루틴이 채널에 값을 쓰거나 읽는 동작에서 동작을 멈추고 해당 채널의 대기열에 멈춘 고루틴을 추가한다. (8장에서 직접 구현해 본다.)

이렇게 완성한 C 프로그램은 10,000번 줄 혹은 1,000,000번 줄도 출력할 수 있다. Java나 JavaScript 혹은 Haskell보다도 훨씬 빠르고 가볍게 실행된다. 역시 C다.

다만 프로그래머가 모든 디테일을 신경써 줘야 한다. `next()`의 일 부분을 함수로 추출하여 재사용하는 것조차 매우 힘들다. App-specific vs. General 관점에서 보자면 일반화 코드가 0이나 마찬가지다.

C #1 - 매크로 푸!

앞 절의 C 코드는 결과를 제대로 얻을 수는 있어도 가독성이 꽝이다. 하지만 C니까 매크로를 적당히 덧씌운다면 가독성을 꽤 획득할 수 있다.[21]

코루틴 함수를 위한 매크로들 (C)

```
#define crBegin    switch(s->ptr) { case 0:
#define crYield(x) do { s->ptr=__LINE__; return x; \
                        case __LINE__:; } while (0)
#define crEnd      } return 0;
```

위 매크로 세 개를 이용하면 `next()`는 아래와 같이 바뀐다. 꽤 읽기 쉽지 않은가? 몇가지 코딩 규칙만 지킨다면 훌륭하게 코루틴을 구현할 수 있다.

매크로를 이용한 코루틴 `next()` (C)

```
int next(proc *s) {
  crBegin;
  crYield(-1);
  s->prev = s->next;
  s->count = 1;
  while(1) {
    crYield(-1);
    if (s->next == 0) {
      break;
    } else if (s->prev == s->next) {
      s->count++;
    } else {
      crYield(s->count);
      crYield(s->prev);
      s->prev = s->next;
      s->count = 1;
    }
  }
  crYield(s->count);
  crYield(s->prev);
  crEnd;
}
```

[21] 이 매크로는 http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html에서 가져온 것을 조금 수정한 것이다.

JavaScript #5 - 제너레이터 코루틴

5장 뒷 부분의 제너레이터 이해하기에서 "제너레이터는 반쪽짜리 코루틴"이라는 표현을 봤었다. 제너레이터는 진입점/탈출점을 여럿 가질 수 있으므로 코루틴이라고 할 수 있다. 다만 차이점은 일반적인 코루틴이 중단하면서 재개할 코루틴을 지정할 수 있는데 반해, 제너레이터는 중단하면 호출한 위치로만 되돌아간다. 이러한 제너레이터 속성을 이용하여 디스패치 함수에서 제너레이터를 선택하여 실행하도록 함으로써 제너레이터를 코루틴처럼 사용할 수도 있다. 두 개 혹은 그 이상의 코루틴을 대상으로 협력적 동시성을 구현하고자 하는 목적이다. 이 절에서는 JavaScript의 제너레이터를 이용하여 코루틴 방식으로 동작하는 개미 수열을 구현하려고 한다.

JavaScript의 제너레이터 객체는 `next()` 메쏘드를 가지는데, 제너레이터를 재개하는 역할을 한다. 이 메쏘드는 제너레이터가 `yield`하는 값을 반환하면서 제너레이터로부터 호출자에 데이터를 전달하는 것과 더불어 `next(value)`처럼 제너레이터를 재개하면서 값을 전달하는 것도 가능하다. 이 경우 제너레이터는 `yield`를 통해 호출자에서 값을 받아오게 된다. (코루틴이라는 이름이 붙을만하다.)

앞서 C로 코루틴의 동작을 구현할 때 구조체를 이용하여 재시작 위치와 지역 변수들을 대신할 상태를 저장해야 했다. JavaScript의 제너레이터는 yield/resume 기능을 제공하므로 재시작 위치를 저장하거나 상태를 따로 저장해 둘 필요가 없다.

제너레이터를 이용한 코루틴 `next()` (JavaScript)

```
function* next() {
  let prev = yield READ
  let count = 1
  let value
  while (value = yield READ) {
    if (prev === value) {
      count++
    } else {
      yield WRITE(count)
      yield WRITE(prev)
      prev = value
      count = 1
    }
  }
  yield WRITE(count)
  yield WRITE(prev)
}
```

5장에서 제너레이터를 이용하여 이터레이터를 만들었을 때와 달라진 점은 `yield`를 사용하는 부분이다. 원래 `yield`는 값을 하나씩 전달하였지만 여기서는 **연산** 혹은 **요청**을 전달한다. `READ`와 `WRITE()`는 아래처럼 정의되었다.

yield 이유를 객체로 반환 (JavaScript)

```
const READ = {type: 'read'}
const WRITE = (value) => ({type: 'write', value})
```

C와 비교하면 훨씬 가독성이 좋아졌다. Go에서 구현했던 next()와 흡사해졌다. Go의 next()에서 in과 out 채널을 이용하여 값을 읽고 쓰고 했던 부분이 yield를 이용하여 값을 주고 받는 것으로 바뀌었다. 이 때문에 일반적인 제너레이터가 이터레이터로 사용되는 것과 달리 코루틴 스타일로 구현된 제너레이터는 코루틴들을 동작시켜 주는 **디스패처**가 필요하다. 디스패치 함수는 코루틴들을 실행하면서 코루틴이 yield하는 연산을 처리한다.

선형적 의존성을 가진 코루틴들을 실행시켜주는 dispatch() (JavaScript)

```
function* dispatch(procs) {
  let value
  let cur = n
  while (true) {
    let next = procs[cur].next(value)
    if (next.done) {
      if (cur === n) {
        return
      } else {
        value = undefined
        cur++
      }
    } else {
      if (next.value.type === 'read') {
        cur--
      } else if (cur === n) {
        yield next.value.value
      } else {
        value = next.value.value
        cur++
      }
    }
  }
}
```

개미 수열의 n번 줄을 출력하는 ant() 함수는 코루틴들을 준비하여 dispatch()를 호출하면 된다. ant()는 다시 일반적인 제너레이터로 래핑되어 있기 때문에 for문에서 사용할 수 있다.

코루틴을 이용하는 **ant()** (JavaScript)

```
function* ant(n) {
  let procs = new Array(n + 1)
  procs[0] = function* () { yield WRITE(1) } ()
  for (let i = 1; i < n+1; i++) {
    procs[i] = next()
  }
  yield* dispatch(procs)
}
```

JavaScript #6 - js-csp

Go처럼 의존 관계를 채널로 명시적으로 드러내고자 한다면 yield에서 채널 객체와 채널 연산(read/write)을 지정해주고, dispatch()에서 해당 연산에 따라 다음 실행될 코루틴을 선택하면 된다. 이를 위해서는 채널 객체마다 read/write 큐를 연관지어 놓아야 한다.

js-csp 라이브러리[22]는 Go의 채널과 고루틴을 구현하고 있으며, 이 라이브러리를 이용하면 Go와 거의 동일하게 구현할 수 있다. js-csp 라이브러리가 제너레이터를 이용할 것이라는 것은 쉽게 예상할 수 있다.

js-csp를 이용하는 **next()** (JavaScript)

```
const {chan, go, put, CLOSED} = require('js-csp');

function next(i) {
  let o = chan()
  go(function* () {
    let prev = yield i
    let count = 1
    for (let value = yield i; value !== CLOSED; value = yield i) {
      if (value === prev) {
        count++
      } else {
        yield put(o, count)
        yield put(o, prev)
        prev = value
        count = 1
      }
    }
    yield put(o, count)
```

[22] https://github.com/ubolonton/js-csp

```
19      yield put(o, prev)
20      o.close()
21    })
22    return o
23  }
```

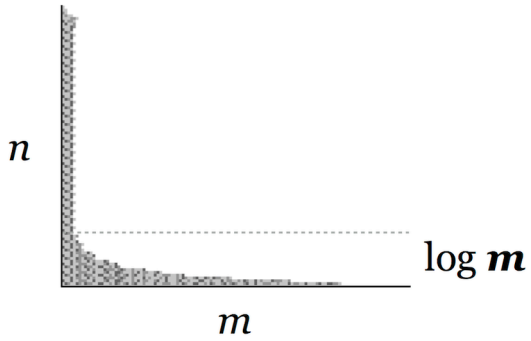
Go의 next()와 비교하면 차이가 없다. JavaScript의 제너레이터를 이용하는 모양은 앞에서 구현한 코루틴과도 거의 비슷하다.

개미 수열 복잡도

이 장에서 C로 구현하면서 모든 추상화를 걷어내고 나니 개미 수열 문제의 복잡도를 계산할 수 있게 되었다. n번 줄의 m번째 글자를 계산하기 위한 복잡도를 따져보자.

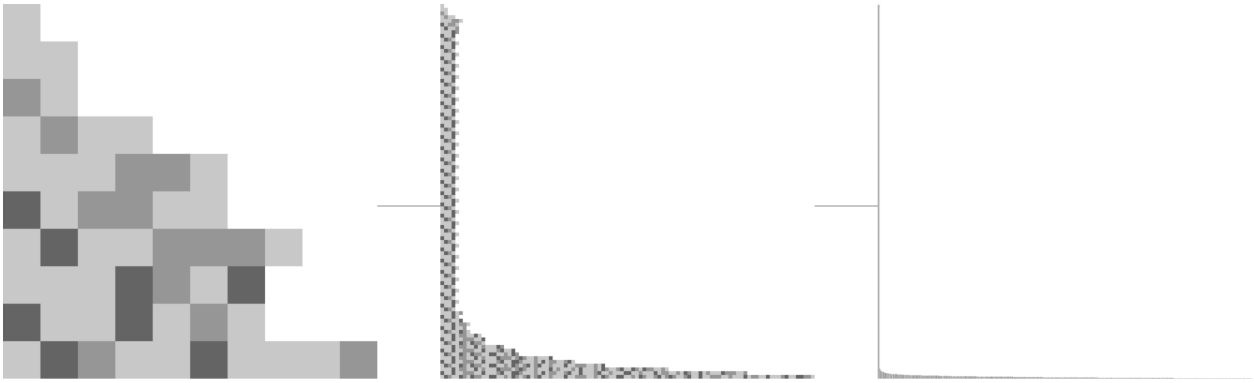
공간 복잡도
: n개의 코루틴이 필요하며, 각 코루틴은 고정된 크기의 메모리만 사용한다. 따라서 O(n)이다.

시간 복잡도
: n번 줄의 m번째 글자를 계산하기 위해서는, n-1번 줄까지는 m보다 적게 계산해도 된다. 정확하게는 m부터 30%씩 줄어드는 등비 수열의 합만큼 계산해야 한다. 줄어드는 등비 수열의 합은 초항 크기에 비례하므로 결국 O(m)임을 알 수 있다. 그런데, m이 줄어들어 최소 너비가 되는 높이는 O(log m)이고 일반적으로 n이 이보다 크다고 보면 높이 n만큼은 탐색을 해야 한다. 따라서 전체 시간 복잡도는 O(n + m)이다.



개미 수열의 복잡도

n과 m이 충분히 클 때, 값을 계산해야 하는 탐색 공간은 가는 L 모양이 되므로 직관적으로 O(n + m)으로 추정할 수도 있을 것이다.



n번째 줄 n번째 글자까지 탐색

코루틴 구현 리뷰

코루틴은 개미 수열 문제의 n이 커졌을 때 발생하는 스택오버플로 혹은 메모리 문제를 해결해 줬다. Go의 고루틴과 채널을 이용하여 개미 수열 문제 혹은 지연 리스트를 동시성 관점에서 바라볼 수 있었다. 이러한 협력적 멀티태스킹을 구현하기 위해 코루틴의 동작(멈춤/재개)을 살펴본 뒤 C와 JavaScript로 구현하여 개미 수열의 n을 1,000,000까지 확대할 수 있었다.

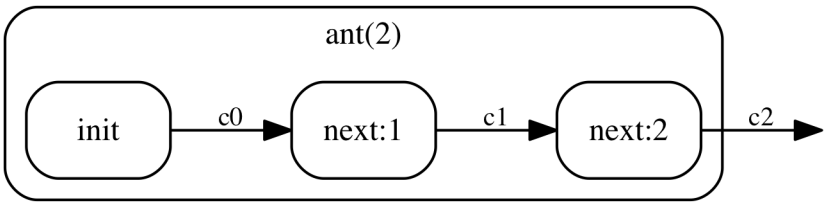
그러나 석연치 않은 구석이 여전히 남았다.

- Go는 다행히 쓰레드보다 가벼운 고루틴이라는 도구를 제공해줬다. 고루틴 같은 도구를 사용할 수 없는 환경에서는 어떻게 할 수 있을까?
- C로 구현했다고는 하지만, 딱 이 문제만 해결할 뿐이다. 추상화 수준이 너무 낮아서 다른 문제에 이러한 방식을 적용하기란 거의 불가능할 것 같다.
- JavaScript는 단일 쓰레드 실행 모델이지만 제너레이터와 js-csp 같은 라이브러리를 이용하여 동시성 프로그래밍을 할 수 있었다. 그러나 여기서도 제너레이터라는 **반쪽짜리 코루틴**이 있었기 때문에 가능했다. Java는 제너레이터를 지원하지 않으며, 5장에서 사용한 Java 제너레이터 라이브러리는 쓰레드를 사용하기 때문에 한계가 분명하다.
- Haskell도 빼먹을 수 없다. Haskell의 그 한 줄은 코루틴과 어떤 관계가 있길래 1,000,000번 줄을 출력할 수 있는 것일까?

8장 CSP와 인터프리터 패턴

6장 코루틴에서 Go의 고루틴이 코루틴과 유사하다는 점을 이야기했다. 사실 Go의 고루틴과 채널의 이론적 뿌리는 CSP(Communcating Sequential Processes) 모델이다. CSP의 핵심은 순차적 실행 단위인 **프로세스**들이 **채널**이라고 하는 통신 매개를 통해서만 동기적으로 메시지를 주고 받는 것으로 동시성을 표현하는 것이다. CSP에서는 프로세스를 "이벤트의 나열로 표현되는 어떤 객체의 행동 패턴"라고 다소 딱딱하게 정의하였지만, 우리는 쉽게, OS 상의 프로세스나 쓰레드, 혹은 Go의 고루틴이라고 보아도 될 것이다. 프로세스는 채널을 통해 다른 프로세스에 메시지(혹은 데이터)를 전달한다.

우리는 이미 7장에서 거의 유사한 모델을 구현해 봤다. 7장에서 구현한 프로세스는 **읽기**와 **쓰기**라는 이벤트들로만 구성되며, 모든 프로세스는 읽기 채널과 쓰기 채널이 하나씩 있는 제한된 형태였다. 아래 그림은 개미 수열의 2번 줄을 출력하는 프로세스 네트워크를 보여준다. 파이프처럼 두 프로세스를 연결하고 c0나 c1이 채널 역할을 하고 있다.



ant(2)에 해당하는 프로세스 네트워크

이번 장에서는 채널을 추가하여 CSP의 동시성 모델을 온전히 지원하도록 확장하고, 이를 이용하여 개미 수열을 구현해 본다. 말하자면 Go의 동작을 따라해 보자는 것이다.

Go의 동시성 요소들

Go에서 지원하는 동시성 요소들은 채널과 고루틴이다. 채널 및 고루틴과 관련된 기본 함수, 연산자, 키워드는 다음과 같다.

`c := make(chan int)`
: `int` 타입의 채널을 만든다. 버퍼 크기를 지정할 수 있으나, 버퍼가 없는 동기 채널이 기본이다.

`c <- 1`
: `c` 채널로 1을 보낸다. 현재 고루틴은 다른 고루틴이 값을 받을 때까지 블록될 수 있다.

`<-c` `c` 채널에서 값을 받는다. 다른 고루틴이 값을 보낼 때까지 블록될 수 있다.

close(c)
: c 채널을 더이상 사용하지 않는 것으로 표시한다. 닫힌 채널에 값을 쓰는 것은 허용하지 않는다.

go f()
: 새로운 고루틴을 만들어서 f()를 실행시킨다.

다섯 개의 연산을 지원한다면 개미 수열을 CSP 모델로 구현하는 데 문제가 없을 것 같다. CPS 구현에서 이미 읽기/쓰기를 구현해 보았고, 특히 Java 버전에서 추가한 Pipe 명령을 참고한다면 go f()를 구현할 수 있을 것이다. 남은 것은 채널이다.

CSP를 위한 미니 언어

Go의 동시성 기능을 참고하여 CSP를 위한 미니 언어를 아래와 같이 정하였다.

- chan(type)
- send(ch, value)
- recv(ch)
- close(ch)
- go(command)
- print(value)
- done(value)
- command.then(v -> command)

앞의 다섯 가지는 이미 설명한 Go의 동시성 요소들과 일대일 매치된다. print(value)는 미니 언어에서 출력을 지원하기 위한 명령이고, done(value)는 프로세스의 계산 결과를 담아 종료하는 명령이다. then()은 프로세스들을 순차적으로 결합시켜 주는 역할을 한다. 7장의 컨티뉴에이션 패싱 스타일을 대신하는 것으로 볼 수 있다.

```
recv(ch, v -> print(v))
// 대신
recv(ch).then(v -> print(v))
```

명령 클래스 구조 (Java)

CSP 미니 언어로 작성된 프로그램은 명령들을 노드로 가지는 트리 구조가 된다. 7장에서 JavaScript는 선형 연결 리스트로 코루틴을 만든 뒤 디스패치는 배열에 담긴 코루틴들을 실행시켰고, Java는 pipe(a, b)라는 명령을 추가하면서 명령의 트리 구조를 만든 바 있다. go() 명령은 pipe()와 비슷하지만, pipe(a,b)는 그 결과로 입출력이 연동된 a와 b 코루틴을 스케줄러에 추가하는 반면, go(a)는 a를 스케줄러에 추가할 뿐 고루틴의 동작은 채널에 의해

서만 의존성이 결정된다. 명령 중에서 chan()과 recv()는 각각 결과 값을 가지므로 동적으로 트리가 확장된다.

CSP 미니 언어의 각 명령에 해당하는 클래스는 명령 실행에 필요한 정보와 다음 명령에 대한 링크를 가진다. 명령들의 공통 부모 클래스를 Command라고 하자. 미니 언어를 위한 생성 메쏘드들을 제공하고 명령 클래스들은 감춘다.

CSP 명령 클래스 구조

```
class Command<A> {
  // 인스턴스 메쏘드
  public <B> Command<B> then(Function<A, Command<B>> f) {
    return new Then(this, f); }

  // DSL을 위한 정적 메쏘드
  public static <A> Command<Channel<A>> chan(Class<A> cls) {
    return new Chan<>(cls, ch -> done(ch)); }
  public static <A> Command<Void> send(Channel<A> ch, A value) {
    return new Send<>(ch, done(null)); }
  public static <A> Command<Optional<A>> recv(Channel<A> ch) {
    return new Receive<>(ch, v -> done(v)); }
  public static <A> Command<Void> close(Channel<A> ch) {
    return new Close<>(ch, done(null)); }
  public static <A> Command<Void> go(Command<A> forked) {
    return new Go<>(forked, done(null)); }
  public static Command<Void> print(String s) {
    return new Print<>(s, done(null)); }
  public static <A> Command<A> done(A value) {
    return new Done<>(value); }

  // 명령 클래스들
  private static class Chan<C,A> extends Command<A> { .. }
  private static class Send<C,A> extends Command<A>{ .. }
  private static class Receive<C,A> extends Command<A> { .. }
  private static class Close<C,A> { .. }
  private static class Go<B,A> extends Command<A> { .. }
  private static class Print<A> extends Command<A> { .. }
  private static class Done<A> extends Command<A> { .. }
  private static class Then<B,A> extends Command<A> { .. }
}
```

send/recv는 7장의 read/write와 거의 같다. 차이 점은 채널 타입을 인자로 받아서 데이터 의존성을 명시적으로 드러낸 것이다. 채널은 chan()으로 생성하고 close()로 닫는다. 동시

실행되는 루틴은 go()로 전달하면 된다. then()에서 컨티뉴에이션을 나중에 연결시킬 수 있기 때문에 처음 노드를 생성할 때에는 항상 기본 값으로 done()을 사용하였다.

핑퐁 예제

클래스 설계를 기초로 간단한 동시성 프로그램을 작성할 수 있다. Go 발표에 몇 번 등장했던 핑퐁 예제[23]를 옮겨보려고 한다. Go 코드로 동작을 먼저 이해하자.

핑퐁 예제 (Go)

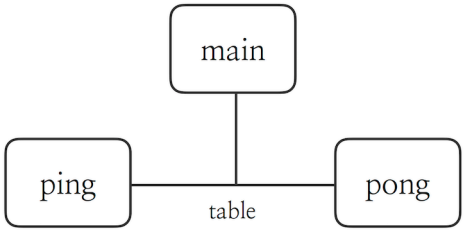
```
1  func main() {
2          table := make(chan *Ball)
3          go player("ping", table)
4          go player("pong", table)
5
6          table <- new(Ball) // 게임 시작, 공을 던진다
7          time.Sleep(1 * time.Second)
8          <-table            // 게임 끝, 공을 가져간다
9  }
10
11 func player(name string, table chan *Ball) {
12         for {
13                 ball := <-table
14                 ball.hits++
15                 fmt.Println(name, ball.hits)
16                 time.Sleep(100 * time.Millisecond)
17                 table <- ball
18         }
19 }
```

이 프로그램은 메인 고루틴에서 "ping"과 "pong" 두 개의 고루틴을 파생시킨다. 세 개의 고루틴은 table 채널을 공유하고, 플레이어 고루틴은 무한 루프를 돌면서 테이블에서 공(Ball)을 받아서 100ms 쉬었다가 다시 보낸다. 두 플레이어 고루틴은 공을 받기 위해서 모두 블록되고, 메인 고루틴이 테이블에 공을 던져 넣으면서 핑퐁이 시작된다. 어느 플레이어가 먼저 공을 칠 것인지는 정해지지 않았다. 1초 뒤에 메인 고루틴이 공을 뺏으면서 종료하면 프로그램이 종료한다.

[23] https://talks.golang.org/2013/advconc.slide#6

CSP 표현

CSP의 프로세스 네트워크를 그릴 때, 프로세스는 박스, 채널은 선으로 표시한다.



핑퐁 예제 프로세스 네트워크

그리고, 프로세스들의 병렬 실행은 `MAIN ¦¦ PING ¦¦ PONG` 처럼, 각 프로세스는 이벤트의 리스트로 표현한다.

```
MAIN = table!ball → sleep → table?ball → STOP
PING = table?ball → sleep → table!ball ' → PING
PONG = table?ball → sleep → table!ball ' → PONG
```

PONG 프로세스는 table 채널에서 값을 읽는 이벤트 다음에 sleep 이베트, 그리고 다시 table 채널에 값을 쓰는 이벤트에 이어서 재귀적으로 PONG 프로세스가 이어진다. ball을 ball ' 로 변환하는 것은 프로세스를 기술하는 입장에서는 중요하지 않으므로 생략했다.

핑퐁 예제에서 main()을 CSP 미니 언어로 옮겨보자.[24]

핑퐁 예제 (Java)

```
1  Command<Void> pingpong() {
2    return chan(Ball.class)
3        .then(table -> go(player("ping", table))
4            .then(x -> go(player("pong", table)))
5            .then(x -> send(table, new Ball()))
6            .then(x -> sleep(1000))
7            .then(x -> recv(table)))
8        .then(x -> done(null));
9  }
```

pingpong() 메쏘드의 반환 타입은 Command<Void>이다. 이는 원래의 Go 프로그램의 main()에 해당한다. 원래의 Go 프로그램의 한 줄마다 Command를 반환하고, 이들을 then()으로 엮어서

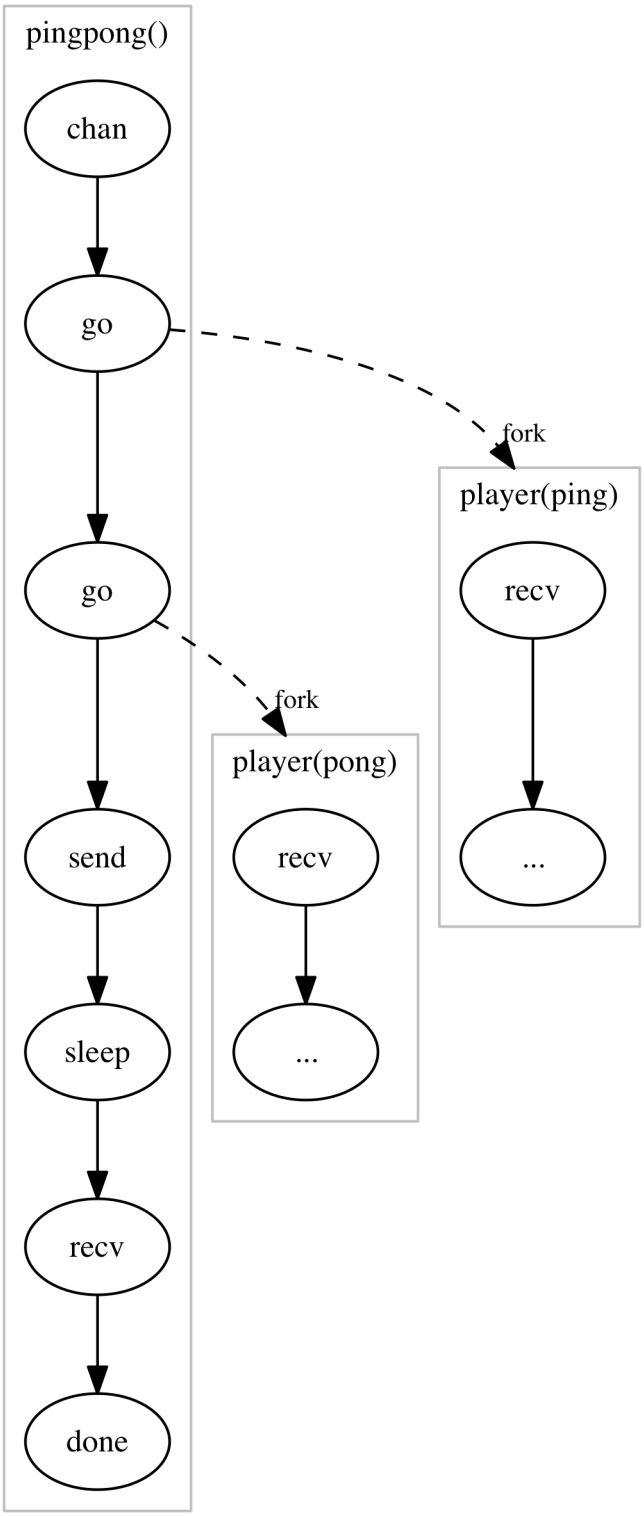
[24] 여기선 sleep() 명령이 추가로 필요하다. 지정된 시간만큼 고루틴의 실행을 멈춘다.

전체 프로그램이 만들어진다. 만약 then() 대신 기존의 콜백 전달 방식을 사용했다면 다음과 같았을 것이다.

then() 없는 핑퐁 예제 (Java)

```
Command<Void> pingpong() {
  return chan(Ball.class, table ->
          go(player("ping", table), () ->
            go(player("pong", table), () ->
              send(table, new Ball(), () ->
                sleep(1000, () ->
                  recv(table, x ->
                    done(null)))))));
}
```

Then 노드를 무시한다면 메인 프로그램의 트리는 아래 그림과 같다.



핑퐁 프로그램의 노드 트리

Java #6 - CSP 개미 수열

6장에서 이미 Go로 구현했던 개미 수열을 CSP 미니 언어로 옮겨보자.

CSP 개미 수열 (Java) – 1

```
  private static Command<Channel<Integer>> ant(int n) {
    return init().then(ch -> loop0(0, n, ch));
  }

  private static Command<Channel<Integer>> init() {
    return chan(int.class)
        .then(ch -> go(send(ch, 1)
            .then(x -> close(ch)))
            .then(x -> done(ch)));
  }

  private static Command<Channel<Integer>> loop0(int i, int n,
                                                 Channel<Integer> ch) {
    if (i < n) {
      return next(ch).then(c -> loop0(i + 1, n, c));
    } else {
      return done(ch);
    }
  }
```

init() 함수는 제너레이터 패턴으로 되어 있고, 자신의 고루틴에서 1을 출력한 뒤 채널을 닫는다. init() 함수의 반환값이 Command<Channel<Integer>>이다. then() 메쏘드로 체이닝이 가능하며, 여기서 채널을 인자로 받아서 연산이 다음 명령을 연결시킨다. 반복문 대신 loop0() 재귀 도움 함수를 이용하여 next()를 n번 호출한다.

CSP 개미 수열 (Java) – 2

```
  private static Command<Channel<Integer>> next(Channel<Integer> i) {
    return chan(int.class)
        .then(o -> go(recv(i)
            .then(c -> loop1(c.get(), 1, i, o))
            .then(x -> close(o)))
            .then(x -> done(o)));
  }

  private static Command<Void> loop1(int prev, int count,
                                     Channel<Integer> i, Channel<Integer> o) {
```

```
    return recv(i)
        .then(value -> {
          if (value.isPresent()) {
            int v = value.get();
            if (v == prev)
              return loop1(prev, count + 1, i, o);
            else
              return send(o, count)
                  .then(x -> send(o, prev))
                  .then(x -> loop1(v, 1, i, o));
          } else {
            return send(o, count)
                .then(x -> send(o, prev));
          }
        });
  }
```

next() 함수 역시 제너레이터 패턴이며, loop1() 재귀 도움 함수를 이용하여 채널의 값을 하나 씩 읽으면서 다음 줄을 계산한다.

도움 함수들

CPS 구현 때와 마찬가지로 기본 반복문을 사용할 수 없다는 점이 아쉽다. 이를 위해 CSP 미니 언어에 반복문을 추가할 수도 있겠지만, 반복문의 패턴을 몇가지 도움 함수로 일반화하여 재사용하는 것이 더 편리하다.

ant()에서 사용된 loop0()은 init()의 결과인 채널을 next()로 n번 반복 호출한다. 어떤 값 a에 함수 f를 n번 거듭 호출하는 것을 applyN(n, f, a)로 일반화할 수 있다.

반복문 도움 함수 – applyN (Java)

```
static Command<Channel<Integer>> ant(int n) {
  return init().then(ch -> applyN(n, LookAndSay::next, ch));
}

static <A> Command<A> applyN(int n, Function<A, Command<A>> f, A initial) {
  if (n == 0)
    return done(initial);
  else
    return f.apply(initial).then(next -> applyN(n - 1, f, next));
}
```

연습. 콜백 스타일로 applyN() 도움 함수 구현하기

이 장에서 도입된 then() 도움 함수는 CPS를 약간 변형한 형태처럼 보일 뿐 필요성이 조금 의심된다. 하지만 applyN()같은 도움 함수를 then()을 사용하지 않고 콜백을 전달하는 방식으로 구현해 보면 그 편의성을 알 수 있다. (JavaScript에서는 크게 차이나지 않는다.)

```
static <A> Command<A> ant(int n, Function<Channel<Integer>, Command<A> k) {
    return init(ch -> applyN(n, LookAndSay::next, ch, k));
}
static ... applyN( ... ) { ... }
```

CSP 미니 언어 인터프리터

CSP 인터프리터는 CSP 명령어 트리를 순회하며 명령을 실행시키면 된다. 실행 중인 프로세스를 리스트로 관리하면서 라운드 로빈 방식으로 **한 번에 하나의 프로세스를 한 단계씩 실행**하면 동시성을 구현할 수 있다.

다음 코드는 채널 생성 명령을 처리한다. 현재 프로세스의 실행할 명령이 Chan인 경우, Channel을 생성하고, 현재 프로세스의 다음 실행할 명령을 계산한 다음 실행 대기열의 끝에 추가한다. 다음 번에 이 프로세스가 실행될 차례가 되면 그 다음 명령이 이어서 실행될 것이다.

CSP 인터프리터의 기본 뼈대 (Java)

```
public class Interpreter {
  public static void run(Command mainCommand) {
    Deque<Process> processes = new LinkedList<>();
    processes.addLast(new Process(true, mainCommand));

    while (!processes.isEmpty()) {
      Process p = processes.removeFirst();
      Command c = p.command;

      if (c instanceof Command.Chan) {
        Command.Chan chan = (Command.Chan) c;
        p.command = (Command) chan.next.apply(new Channel(chan.cls));
        processes.addLast(p);
      } else {
        ...
      }
    }
```

```
18    }
19  }
```

CSP 미니 언어는 Java 제너릭스를 이용하여 타입 안정성을 확보하였지만, 인터프리터는 런타임 타입 정보를 이용할 수 없기 때문에 rawtype으로 구현하였다.

send()/recv() 처리하기

send(ch, value) 명령은 채널에서 값을 받기 위해 대기 중인 프로세스가 있는 경우 대기 중인 프로세스를 깨우면서 값을 전달하고, 그렇지 않은 경우에는 현재 프로세스를 **보내기 대기열**에 추가한다. recv(ch) 명령은 반대로 동작하여 채널에 보내기 대기 중인 프로세스가 있으면 대기 중인 프로세스를 깨우면서 값을 전달 받고, 그렇지 않으면 현재 프로세스를 **받기 대기열**에 추가한다.

이를 위해 Channel은 보내기/받기 각각의 프로세스 리스트를 가진다. 이 리스트에 추가되는 프로세스는 블록되어 실행이 멈추고, 값의 보내기/받기가 동기적으로 이뤄지면서 다시 실행 대기열에 들어간다.

send() 명령 처리 (Java)

```java
if (c instanceof Command.Send) {
  Command.Send send = (Command.Send) c;
  Channel ch = send.ch;
  if (ch.isClosed) {
    throw new RuntimeException("Send to a closed channel");
  } else if (ch.receivers.isEmpty()) {
    ch.senders.add(p); // 현재 프로세스를 보내기 대기열에 추가한다.
  } else {
    // 받기 대기 중인 프로세스를 깨운다.
    Process receiver = (Process) removeRandom(ch.receivers);
    Command.Recv recv = (Command.Recv) receiver.command;
    receiver.command = (Command) recv.next.apply(Optional.of(send.value));
    processes.addLast(receiver);

    p.command = send.next;
    processes.addLast(p);
  }
}
```

recv()는 send()와 거의 비슷하게 구현할 수 있다. 차이점은 채널이 닫힌 경우에 대한 처리이다. Go의 스펙을 참고하면, 닫힌 채널에 값을 보내는 경우 **패닉**(panic)을 발생시키고, 닫힌

채널에서 값을 받으려 하면 **제로 값**을 받는 것으로 되어 있다. 여기서는 제로 값을 대신하여 `Optional.empty()`를 사용하기로 했다.

마지막 줄의 `processes.addLast()`를 `addFirst()`로 바꾼다면 어떻게 될까? 라운드 로빈 방식이 아니게 되지만, 동작에는 차이가 없다. 새로 깨어난 프로세스를 실행 대기열의 맨 앞에 추가하더라도 동작에 문제가 없다.

go() 처리하기

동시성 지원 명령 중에서 `go()`는 새로운 프로세스를 생성하여 실행 프로세스 대기열에 추가한다. 물론 현재 진행 중인 프로세스도 계속 진행되어야 하므로 실행 대기열에 추가해야 한다.

go() 명령 처리 (Java)

```
if (c instanceof Command.Go) {
  Command.Go go = (Command.Go) c;
  processes.addLast(new Process(false, go.forked));
  p.command = go.next;
  processes.addLast(p);
}
```

done() 처리하기

이쯤에서 `Process` 클래스를 살펴봐야겠다. 생성자에서 `Command`뿐만 아니라 불린 값을 하나 받는다. 이 불린 값은 이 프로세스가 **메인** 프로세스인지 여부를 나타낸다. 메인 프로세스를 표시해 두는 이유는 다른 프로세스들과 달리 메인 프로세스가 종료되었을 때 전체 실행을 중단하기 위해서다.

Process 클래스 (Java)

```
public class Interpreter {
  ...
  static class Process {
    final boolean isMain;
    Command command; // 현재 프로세스에서 실행할 명령 포인터

    Process(boolean isMain, Command command) {
      this.isMain = isMain;
      this.command = command;
    }
  }
}
```

이제 `done()`을 처리하는 것은 간단하다. 메인 프로세스가 종료되면 전체 실행 루프를 끝낸다. 다른 프로세스가 종료되면 실행 대기열에 더 이상 추가하지 않으면 된다.

done() 명령 처리 (Java)

```
if (c instanceof Command.Done) {
  if (p.isMain) {
    break;
  }
}
```

then() 처리하기

CSP 미니 언어에서 프로세스는 명령의 순차적 리스트이다. 여기서 then()의 역할은 두 개의 명령을 이어 붙여서 (심지어 done()에 대해서도) 일반적인 프로그램의 함수 호출과 같은 구조화를 가능하게 한다.

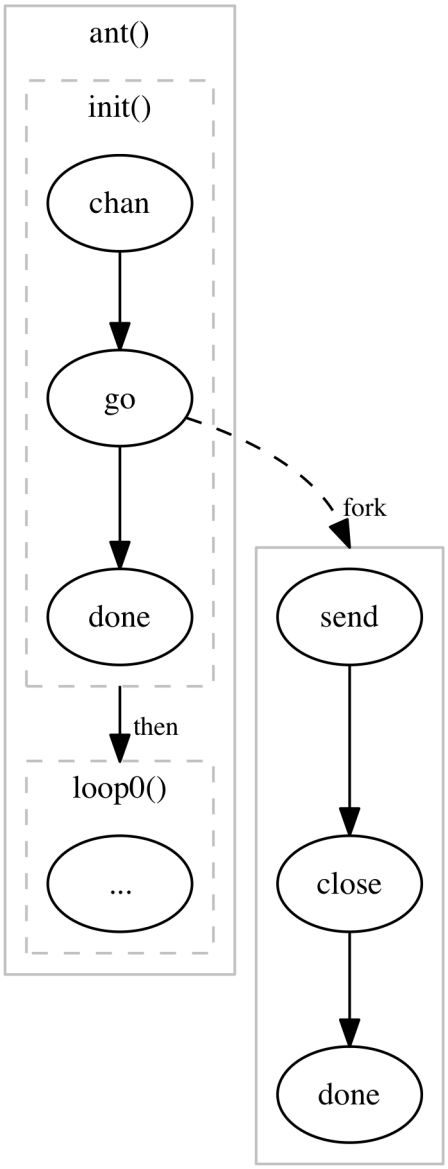
Then 명령 클래스 (Java)

```
class Command<A> {
  <B> Command<B> then(Function<A, Command<B>> f) {
    return new Then<>(this, f);
  }
}

class Then<A, B> extends Command<B> {
  final Command<A> sub;
  final Function<A, Command<B>> k;

  Then(Command<A> sub, Function<A, Command<B>> k) {
    this.sub = sub;
    this.k = k;
  }

  @Override
  public <C> Command<C> then(Function<B, Command<C>> f) {
    return new Then<>(sub, a -> k.apply(a).then(f));
  }
}
```

Then 클래스는 then() 메쏘드를 오버라이드하는데, 그 이유는 불필요한 중첩을 사전에 막기 위한 것이다.

앞서 미니 언어로 구현한 개미 수열의 ant() 함수는 다른 함수 init()을 호출하는데, 이 함수는 done()으로 종료하는 작은 명령 트리를 반환한다. done()은 원래 프로세스가 종료되고 실행할

다음 명령이 없는 것을 의미하지만, ant()에서는 마치 **서브 루틴**을 호출한 것처럼 then()으로 다음 동작을 이어간다.



ant()에서 init()과 loop0()을 호출하는 모양

즉, then() 명령으로 생성되는 노드는 실제 어떤 행위를 나타낸다기보다는 리스트 두 개를 이어주는 역할을 할 뿐이므로 인터프리터는 다음 실행할 명령을 얻어올 때 then() 노드를 모두 건너뛰어야 한다. (만약 Then.sub를 run()으로 실행하여 결과를 받아와서 Then.k를 실행시킨다면 콜스택을 사용하는 문제가 다시 생겨난다.)

Interpreter는 then()을 스킵 (Java)

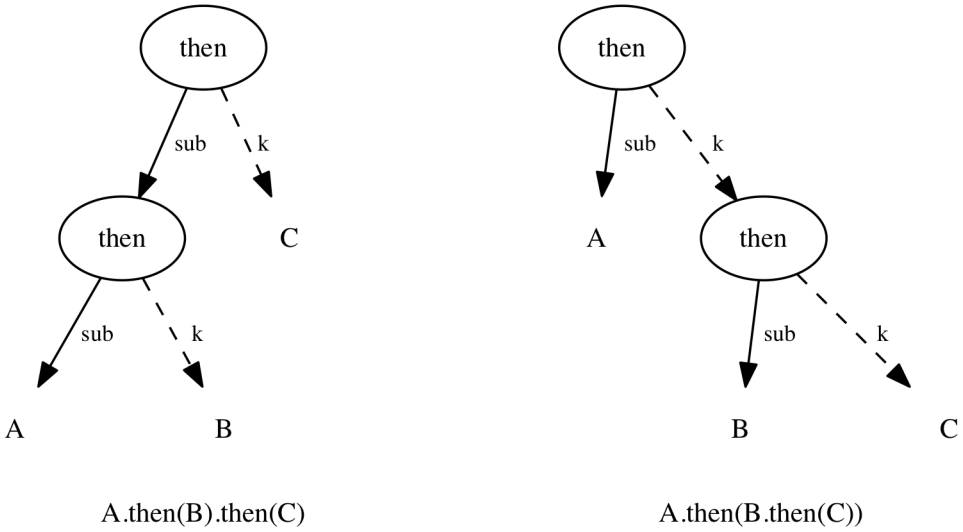
```java
public static void run(Command mainCommand) {
  Deque<Process> processes = new LinkedList<>();
  processes.addLast(new Process(true, mainCommand));

  while (!processes.isEmpty()) {
    Process p = processes.removeFirst();
    Command c = next(p.command);
    ...
  }
}

private static Command next(Command command) {
  Command c = command;
  while (c instanceof Command.Then) { // `Then`은 모두 건너뛴다.
    Command.Then then = (Command.Then) c;
    if (then.sub instanceof Command.Done) {
      Command.Done done = (Command.Done) then.sub;
      c = (Command) then.k.apply(done.value);
    }
    ...
  }
  return c;
}
```

여기서 특이한 케이스는 then()에 의해 선행 실행되어야 하는 명령이 다시 then()인 경우다. 이건 마치 서브 루틴을 중첩 호출한 것과 같아서 다음 명령 찾기 루프를 계속 진행해야 한다. 이때, 콜스택에 해당하는 컨티뉴에이션 k가 누적되지 않도록 중첩된 컨티뉴에이션(sub.k) 다음으로 이어주어야 한다.

중첩 then() 건너뛰기 (Java)

```java
while (c instanceof Command.Then) { // `Then`은 모두 건너뛴다.
  Command.Then then = (Command.Then) c;
  if (... Done) {
    ...
  } else if (then.sub instanceof Command.Then) { // nested Then
    Command.Then sub = (Command.Then) then.sub;
    c = sub.sub.then(a -> ((Command) sub.k.apply(a)).then(then.k));
  }
```

조금 복잡한 7번 줄을 그림으로 옮겨보면 다음과 같다. A, B, C의 실행 순서를 지키면서 then() 중첩을 컨티뉴에이션 쪽으로 옮긴다. 다음 루프를 통해 A를 탐색한다.



중첩된 then() 처리하기

CSP 인터프리터 리뷰

CSP는 동시에 진행되는 여러 프로세스들의 동작을 기술할 수 있는 언어와 동작을 검증할 수 있는 수학적 토대를 제공한다. 쓰레드와 락 중심의 동시성 프로그래밍 대신 동기화 수단으로서 채널을 이용하는 방법은 동시성 프로그램을 새롭게 설계할 수 있게 도와준다.

이 장에서는 Go를 본보기로 하여 CSP 미니 언어를 디자인하고, 인터프리터를 만들어 보았다. 더이상 개미 수열의 1,000,000번 줄을 출력하는 것은 문제가 되지 않는다. 멀티 쓰레드를 사용하지 않지만 코루틴의 메커니즘으로 동시성 프로그래밍까지 가능하다는 것을 확인했다.

하지만 이 장에서 구현한 CSP 미니 언어는 아직 제약이 많다. Go와 비교해 보면 아래의 몇 가지 기본 기능이 빠져있다.

1. 버퍼 채널 – 버퍼가 지정된 채널은 버퍼 크기만큼 보내기만 먼저 발생할 수 있다.
2. `sleep()` – 프로세스를 일정 시간만큼 잠 재운다.
3. `select()` – 두 개 이상의 채널 연산 중 먼저 발생하는 이벤트로 흐름을 진행한다.

연습. CSP 미니 언어에 sleep() 추가하기

이 장의 핑퐁 예제는 sleep()이 필요하다. 예제가 동작하도록 sleep() 명령을 추가해 보자.

```
Command<Void> sleep(long ms) { ... }
class Sleep exends Command<A> { ... }
```

연습. CSP 미니 언어에 select() 추가하기

6장에서 예제로 보여준 Go 프로그램은 boring() 제너레이터에서 값을 다섯 개 읽는다. 이 예제를 확장한 아래의 Go 프로그램[25]은 boring()에서 보내는 메시지를 수신할 때 1초가 지나면 종료한다. CSP 미니 언어에 select()를 추가하고 아래 프로그램을 구현해 보자.

```
1  func main() {
2      c := boring( " Joe " ) // 2초 이내 랜덤하게 메시지 발생
3      for {
4          select {
5          case s := <-c:
6              fmt.Println(s)
7          case <-time.After(1 * time.Second): // 1초 타임아웃
8              fmt.Println( " You ' re too slow. " )
9              return
10         }
11     }
12 }
```

코루틴이라는 아이디어에서 출발하여 여러가지 방식으로 개미 수열을 풀어보았다. 하지만 Go처럼 런타임 스케줄러가 없는 경우, 즉 우리가 코루틴을 흉내낸 경우는 한결같이 디스패쳐가 필요했다. 아이디어를 확인하고 연습하기에는 적절하지만 실용적이라고 하기는 어려울 것 같다.

디스패쳐는 명령 노드들의 느슨한 연결 리스트를 하나씩 따라가는 역할을 한다. 이 부분에서 우리는 **느슨한 연결 리스트**라는 새로운 도구를 얻었다. 이를 일반화하여 지연 리스트를 만들 수 있다. 9장에서 다룰 내용이다.

[25] https://talks.golang.org/2012/concurrency.slide#35