**두 개의 관점**

저는 어떤 개념을 공부할 때, 크게 두 가지 관점에서 그 개념을 살펴봅니다.

1. 이것의 정의는 무엇인가? — 개념의 겉뜻, 겉모습 탐구
2. 이것이 의미하는 바는 무엇인가? — 개념의 속뜻, 시사하는 바 탐구

개념의 겉뜻을 모르면 개념을 사용할 수 없습니다. 마치 그림을 못 그리는 사람들이 그림 그릴 때 처럼요. 머릿속에서는 온갖 상상이 펼쳐지지만 손은 그걸 도화지에 투영해내지 못합니다. 개념의 겉뜻을 모르는 경우도 마찬가지입니다. 개념을 어떻게 활용할 지에 대한 대략적인 형태와 배경은 존재하나 이를 직접 실제적인 것으로 투영해내지는 못합니다. 반대로 개념의 속뜻을 모르면 개념을 활용할 수 없습니다. 마치 수학 심화 문제를 풀 때 처럼요. 문제를 읽어 보면 무슨 기술을 써야 할 지는 알 거 같은데, 그걸 어떻게 써야 할 지 도통 알 수 없는 것입니다. 개념의 속뜻을 모르는 경우도 마찬가지입니다. 무엇이 그 개념에 해당하는지는 알지만, 그 개념을 가지고 사고를 발전시키지는 못합니다.

이렇게, **개념의 겉뜻과 속뜻은 처음에는 정말 상이해보이지만 신기하게도 결국엔 하나로 합쳐져 온전한 개념을 이룹니다**. 그렇기에 저는 어떠한 개념을 이해함에 있어 두 관점에서의 관찰이 이루어지지 않으면 안 된다고 생각하며, 이 글 또한 마찬가지로 두 개의 관점에서 모나드를 소개하는 형태로 써 보려고 합니다.

**첫 번째 관점에서의 관찰, 모나드의 정의는 무엇인가?**

**어떤 타입 M에 대해 아래의 두 함수, pure과 compose가 존재할 때, M은 모나드입니다.**

type Pure = <A>(a: A) => M<A>;

type Compose = <A, B, C>(f: (a: A) => M<B>, g: (a: B) => M<C>) => (a: A) => M<C>;

네, 이게 프로그래밍 세계에서의 모나드의 정의입니다. 생각보다 많이 간단하지요? 우리에게 친숙한 프로그래밍 언어로, 프로그래밍 세계라는 수학보다 훨씬 한정적인 범위 내에서 정의해서 그렇습니다. 우리가 알고 싶은 건 프로그래밍 세계에서의 모나드지, 수학에서의 모나드가 아니니까요. 프로그래밍 세계에서의 정의로도 충분하지요.

이렇게 정의를 하나 보여드렸으니, 이번에는 구체적인 사례를 보여드리겠습니다. 아래는 Maybe라는 모나드로, 모나드 중에서 간단한 축에 속하는 녀석입니다.

type Maybe<A> = A | null;

function pure<A>(value: A): Maybe<A> {

return value;

}

function compose<A, B, C>(f: (a: A) => Maybe<B>, g: (a: B) => Maybe<C>): (a: A) => Maybe<C> {

return (a: A): Maybe<C> => {

const ma = f(a);

if (ma === null) return null;

else g(ma);

}

}

지금 기분이 알 듯 말 듯, 반쪽만 배운 기분이시라면, “그래서 이게 뭐? 무엇이 모나드인지는 알겠는데 이게 프로그래밍에서 갖는 의미가 뭔데?” 싶으시다면 여러분은 정상이십니다! 왜냐하면 제가 아직 모나드의 반쪽만 설명해드렸기 때문이죠. 반쪽만 듣고 전체를 다 알아내는 사람이 있으면 그건 그 분이 뛰어나신 겁니다. 원래 반쪽만 들어선 모르는 게 정상이에요. 오히려 제가 설명한 반쪽, 겉뜻을 완전히 이해하신 거니 잘 하신 겁니다. 그럼 이제 나머지 반쪽, 속뜻에 대해서 이야기 해 보도록 하지요.

**두 번째 관점에서의 관찰, 모나드가 의미하는 바는 무엇인가?**

우리는 프로그램을 작성할 때, 어떤 연산이 존재하고, 어떤 데이터가 어떤 연산으로 처리되어야 할 지를 서술하곤 합니다. 그렇기에 프로그램은 ‘연산의 명세’라고도 볼 수 있습니다. 이 관점에서 **연산은 프로그램을 구성하는 뼈대로, 프로그래밍에 있어 빼놓기 어려운 매우 중요한 요소**라고 할 수 있겠습니다.

하지만, 안타깝게도 프로그래머들은 보통 값을 우대하곤 합니다. 그말인 즉슨, 값을 어떻게 잘 모델링할 지를, 추상화 할 지를 고민하지 연산을 어떻게 모델링 하고 취급할 지는 잘 고민하지 않는다는 말입니다. 일반적으로, 단순한 함수를 통해 연산을 어떻게 표현할 지를 제외하면 심도 깊은 고민들을 잘 하지 않죠. 하지만 프로그램을 ‘연산의 명세’로 보는 관점에 따르면 연산은 데이터만큼이나 프로그래밍에 있어 중요한 요소입니다. 그러니 연산들에 대해 이야기 해 봅시다.

추상화의 한 방식으로, **분류(*classification*)**란 방식이 존재합니다. **분류란 말 그대로 공통적인 속성을 갖는 구체적인 사례들을 분류하여 추상적인 개념을 만들어 내는 방식**입니다. 예컨대, 프로그래밍 언어들의 모음이 있을 때 ‘자바’, ‘C++’, ‘자바스크립트’와 같이 ‘객체’를 중심적으로 다루는 언어들을 따로 묶음으로서 ‘객체 지향 프로그래밍 언어’라는 하나의 추상적인 개념을 만들 수 있지요. 이렇게 공통적인 속성을 갖는 구체적인 사례들을 분류하여 새로운 카테고리를 만드는 일, 그것이 분류입니다. 그럼 이번에는 연산들을 분류해 봅시다. 정말 다양한 분류가 나올 수 있을 텐데요, 그 중에서 저는 ‘나눗셈’, ‘배열에서 첫 번째 요소 가져오기’라는 ‘예외를 발생시키는 경우가 존재하는 연산’들을 따로 묶어서 ‘실패할 수 있는 연산’이라는 하나의 추상적인 개념을 만들어 보았습니다.

이렇게 연산들을 분류하고 보니 실패할 수 있는 연산에 무엇이 있는지는 알 수 있을 것 같습니다. 그런데 이걸 어떻게 표현할 수 있을까요? ‘실패할 수 있는 연산’은 구체적으로 어떻게 정의해야 하는 걸까요? 단순히 ‘나눗셈과 같은 것, 배열에서 첫 번째 요소를 가져오는 것과 같은 것’이라고 할 수는 없잖아요. 이런 미약한, 느슨한 정의로는 어떤 연산이 실패할 수 있는 연산이고, 어떤 연산이 아닌 지 구별하기가 어렵습니다. 좀 더 **명확하고 구체적인 언어**가 필요합니다.

다행히도, 우리가 사용하는 프로그래밍 언어들은 이런 미약한 방식으로 연산들을 정의하지 않습니다. 대신, **의미론(*semantic*)이라는 것을 활용해 연산에 대한 충분히 명확하고 구체적인 정의**를 내리지요. 연산 의미론(*operational semantic*), 표기 의미론(*denotational semantic*), 공리 의미론(*axiomatic semantic*) 등 정말 다양한 종류의 의미 구조가 존재하는데요, 그 중 우리가 주목해야 할 것은 바로 범주 의미론(*categorical semantic*)입니다.

범주 의미론은 쉽게 말해 카테고리 이론(*category theory*, a.k.a 범주론)의 개념들로 연산을 정의하는 방식입니다. 카테고리 이론은 집합론의 반대에 해당하는 개념으로 ‘요소’ 대신 ‘요소들 간의 관계’에 주목하여 추상적인 개념들을 다루고자 하는 수학 이론인데요, 이것이 무엇인지는 당장 크게 중요하지 않으니 더 이상 이야기 하지 않겠습니다. 중요한 것은 모나드가 바로 이 이론의 개념이라는 점과, 유지니오 모기(*Eugenio Moggi*)라는 컴퓨터 과학자가 범주 의미론에서 **모나드로 연산을 정의하고 추상화 할 수 있다는 걸 발견했다는 점**입니다.

그렇습니다. 프로그래밍에서 **모나드는 연산을 정의하고, 추상화 하기 위해 쓰이는 것**입니다. 하지만 아직 의문점은 남아있습니다. 어떤 연산을 정의한다는 것이며 모나드로 연산을 정의하면 무엇이 좋길래 함수형 프로그래머들은 이걸 그리도 좋아하는 걸까요? 먼저, 연산을 왜 추상화 해서 다루는 지를 생각해 봅시다.

제네릭 프로그래밍(*generic programming*)을 아시나요? **제네릭 프로그래밍은 여러 데이터들에 대해 일반적인, 즉, 유연한 코드를 작성함으로서 재사용성을 끌어올리는 방식**입니다.

class List<T> {

public readonly head: T;

public readonly tail?: List<T>;

constructor(head: T, tail?: List<T>) {

this.head = head;

this.tail = tail;

}

}

모든 타입에 대해, 각 타입의 리스트는 담고 있는 값은 다를지언정 리스트 자체의 구조는 항상 가장 앞 부분과 뒷 부분이 있다는 점에서 같지요. 위의 코드는 이를 이용한 제네릭 프로그래밍의 예 입니다.

그렇다면, 만약 데이터들 뿐만 아니라 연산마저 추상화를 한다면 어떨까요? 코드의 재사용성이 더 증폭되겠지요? 이것이 바로 연산을 추상화하는 이유입니다. 그리고 이 ‘연산 추상화’에 있어, 어떤 연산이 범주 의미론에서 모나드로 추상화 된다면 그 연산은 ‘합칠 수 있음’이 보장됩니다. 수학적으로, 모나드의 성질이 바로 ‘합칠 수 있음’이기 때문이죠. 즉, **모나드는 ‘합칠 수 있는 연산’을 정의하고 추상화 하기 위해 쓴다**고 할 수 있겠습니다. 그럼 여기서 또 다른 의문점이 생깁니다. 연산이 합쳐질 수 있다는 것은 어떤 의미를, 그리고 가치를 갖는 걸까요?

프로시저, 서브루틴, 함수, 익숙한 이름이죠? 그렇다면 부프로그램(*subprogram*)은 들어 보셨나요? 이 이름들은 모두 연산을 추상화 할 때 쓰이는 개념의 이름입니다. 조금씩 세부 사항이 다를 뿐, 근본은 모두 같지요. 그런데 잠시만요, 지금 부 **프로그램**이라고요? 그렇습니다. 사실 함수들은 부프로그램, 다시 말해 프로그램 속의 프로그램입니다. 그리고 함수는 연산을 추상화 하는, 연산을 나타내는 개념이지요. 그말인 즉슨, **‘모든 프로그램은 곧 연산’이라는 말입니다! 그리고 모나드로 정의되는 연산은 합쳐질 수 있습니다!** 이제 모나드의 매력을 아시겠나요?

정리해 보지요.

어떤 연산이 모나드로 정의된다면, 그 연산은 합칠 수 있습니다. 프로그램은 곧 연산이고, 연산은 곧 프로그램입니다. 그렇기에 **어떤 연산이 모나드로 정의된다면, 그 연산에 해당하는 모든 프로그램은 합쳐질 수 있습니다!** (일례로, 입출력 연산은 모나드로 정의될 수 있습니다. 그렇기에 하스켈 프로그래머들은 입출력 프로그램들을 합치며 프로그램을 만들지요.)

지금까지의 모든 이야기를 종합하자면 모나드의 속뜻, 즉, **어떤 것이 모나드임이 시사하는 바는 그것이 합성될 수 있는 연산**이라는 겁니다.

**두 관점 합치기, 모나드란 무엇인가?**

지금까지 각 관점에서 모나드를 관찰해 보았습니다. 마지막으로 각 관점에서 관찰한 것들을 반대 관점에서 얻은 지식으로 보충하여, 두 관점에서 깨달은 내용들을 합치고 모나드의 전체적인 모습을 그려 보도록 하겠습니다.

**속뜻에서 겉뜻으로, 모나드의 정의는 이렇게 만들어졌다.**

모나드의 정의를 다시 떠올려 봅시다.

***어떤 타입 M에 대해 두 함수, pure과 compose가 존재할 때, M은 모나드다.***

제가 처음 이 정의를 제시했을 때에는 아직 모나드에 대한 충분한 논의가 이루어지지 않았었기 때문에 외적인 요소들만 소개해드리는 데에 그쳤었습니다. 그러나 지금은 이 정의의 각 부분이 무엇을 의미하는지 탐구할 수 있을 정도로 이야기가 진행되었기 때문에 지금부터는 정의를 분해해보며 각 부분이 어떤 의미를 담고 있는지 알아보도록 하겠습니다.

먼저, 타입 M부터 살펴보도록 하겠습니다. **타입 M은 연산 M을 의미하며 타입 M<A>는 A 타입의 연산 M을 의미합니다.** 예컨대, M<number>는 number 타입의 값을 만드는 연산 M을 의미하지요. 그런데 타입 M은 연산과 어떤 관계길래 연산을 나타낸다고 하는 걸까요? 아까 보았었던 Maybe 타입의 구현도 그렇고, 지금 알아본 M 타입의 의미 또한 전혀 함수와 어떤 관련도 있어 보이지 않는데 말이죠.

M 타입이 연산을 대표하는 이유는 모든 프로그램을 A 타입의 값을 받아 연산 M을 수행하여 B 타입의 값을 내놓는 함수라고 볼 수 있다는 점에서 찾을 수 있습니다. 마치 모든 일차방정식이 ax + b = 0 꼴로 표현될 수 있는 것처럼, 모든 프로그램 또한 A => M<B> 타입으로 표현될 수 있는데요. 여기서 **타입 A와 B는 항상 중복되기 때문에 각 프로그램의 차이는 연산을 나타내는 타입 M으로부터 비롯됨**을 알 수 있으며, 이는 ‘프로그램은 연산의 명세’ 라는 직관과도 일치합니다. 그렇기에 타입 M을 두고 연산 M을 의미한다고 하는 것입니다.

다음으로는 compose 함수의 의미에 대해 알아보도록 하겠습니다. compose 함수의 시그너쳐는 <A, B, C>(f: (a: A) => M<B>, g: (a: B) => M<C>) => (a: A) => M<C>로, 같은 연산을 하는 두 프로그램을 받아 하나로 합치는 형태이며 실제 동작도 마찬가지입니다. 모나드가 ‘합성할 수 있는’ 연산을 정의하고 추상화하는 도구라는 점, 잊지 않으셨지요? 바로 이 함수가 모나드인 연산의 합성 가능성을 보장하는 함수입니다. 모나드가 되려면 이 함수를 구현해야 하기에, 모나드인 모든 연산들이 합성될 수 있다고 하는 것이지요. 즉, **compose 함수는 합성 가능한 연산들만이 모나드가 될 수 있게 하기 위해 존재하는 제한**인 셈입니다.

마지막으로, pure 함수의 의미에 대해 알아보도록 하겠습니다. **pure 함수는 일반적인 값을 연산 M이 적용된 값인 것 처럼 꾸며 주는 함수**입니다. 예컨대, pure(1)은 1을 결과로 하는 연산 M을 나타내지요. 이 함수의 쓸모에 대해 이야기하려면, 먼저 id 라고 불리는 함수에 대해 알아야 합니다.

id 함수란, 주어진 입력을 그대로 반환하는 함수로. 아래와 같이 구현될 수 있습니다.

function id<A>(a: A): A {

return a;

}

놀랍도록 간단한 이 함수는, 함수형 프로그래밍에 있어 가장 중요한 함수 중 하나입니다. 이론적으로도, 실용적으로도 많은 면에서 다양하게 쓰이지만, 가장 쉬운 사용례로는 **함수의 몸집 키우기**가 있겠습니다.

예컨대, 임의의 튜플 [A, B]의 각 요소에 함수를 적용하는 함수, bimap이 있다 가정합시다.

function bimap<A, B, C, D>([a, b]: [A, B], f: (a: A) => C, g: (b: B) => D): [C, D] {

return [f(a), g(b)];

}

그리고 주어진 튜플의 두 번째 요소에만 함수를 적용하는 함수를 만들어야 한다고 한다면. id 함수 없이는 아래와 같이 부분적으로 중복되는 코드를 처음부터 작성해야 합니다.

function fmap<A, B, C>([a, b]: [A, B], f: (b: B) => C): [A, C] {

return [a, f(b)];

}

하지만 id 함수와 함께라면 정말 간단하게 필요한 함수를 만들 수 있습니다.

function fmap2<A, B, C>(pair: [A, B], f: (b: B) => C): [A, C] {

return bimap(pair, id, f);

}

이렇게, id 함수는 작은 함수들을 조립해 큰 함수를 만들 때, 별다른 기능 없이 단순히 특정 자리를 채워 몸집만을 불리고 싶은 경우에 사용할 수 있습니다. 하지만 이 함수는 A => M<B> 꼴이 아니기 때문에 모나드인 연산들을 다루는 경우에는 쉽게 사용할 수 없다는 문제점이 있습니다. (예컨대, compose 함수는 id 함수를 인자로 받지 않지요.) 이를 해결하기 위한 것이 바로 pure 함수, **연산 M에 대한 id 함수**입니다.

**겉뜻에서 속뜻으로, 모나드는 이렇게 쓰인다.**

이제 모나드의 정의와 정의가 의미하는 바를 알았으니 마지막으로 직접 모나드인 연산과 모나드인 연산들을 다루는 함수들을 정의하며 모나드의 가치를 체험해보도록 하겠습니다.

먼저, 모나드인 연산부터 정의해 보도록 하겠습니다. 이번에 정의해 볼 연산은 입출력 연산입니다. **입출력 연산은 외부 세계를 조작하는 연산**으로, 외부 세계를 나타내는 값의 타입 RealWorld와 입출력 연산 결과의 타입 A를 통해, 외부 세계를 받아 수정된 외부 세계와 결과값을 내놓는 함수들의 타입, RealWorld => [A, RealWorld]으로 표현될 수 있습니다. 가장 먼저, 입출력 연산을 나타낼 타입부터 정의해 보지요.

/\*\*

RealWorld 타입은 입출력 연산을 '의미상 순수하게' 취급하기 위한 트릭키한 타입입니다.

그렇기에 RealWorld 타입은 일반적으로 원시 타입으로 정의되나, 타입스크립트에는 존재하지 않는 타입이므로 지금은 그냥 never 타입으로 대체하도록 하겠습니다.

\*\*/

type RealWorld = never;

type IO<A> = (realWorld: RealWorld) => [A, RealWorld];

다음으로, pure 함수를 정의해 보도록 하겠습니다.

function pure<A>(a: A): IO<A> {

return realWorld => [a, realWorld];

}

간단하지요? pure는 어떤 값 a를 받아 외부 세계에 아무런 조작도 하지 않고 그저 a만을 결과로 하는 입출력 연산을 내놓는 함수이므로, 주어진 값을 그대로 반환하는 id 함수와 근본적인 아이디어가 같다고 할 수 있겠습니다. 그럼 마지막으로 두 입출력 연산을 합쳐 줄 compose 함수를 정의하도록 하겠습니다.

function compose<A, B, C>(f: (a: A) => IO<B>, g: (a: B) => IO<C>): (a: A) => IO<C> {

return a => realWorld => {

const [prevResult, prevRealWorld] = f(a)(realWorld);

return g(prevResult)(prevRealWorld);

}

}

이렇게 입출력 연산을 정의해 보았습니다. 다음으로, 모나드인 연산들을 다루는 함수를 하나 정의해 보도록 하겠습니다. 이번에 정의할 함수는 bind라고 불리는 함수로, 연산의 결과에 연산을 적용하는 함수입니다.

/\*\*

타입스크립트에서, HKT(제네릭 타입)은 타입 변수로 전달받을 수 없기 때문에 실제로는 타입 레벨에서 트릭을 사용해 전달하곤 합니다.

그러나 지금은 주제에 집중하기 위해 HKT가 타입 변수로 전달될 수 있다고 가정하고 작성하겠습니다.

\*\*/

interface Monad<M> {

pure<A>(a: A): M<A>;

compose<A, B, C>(f: (a: A) => M<B>, g: (a: B) => M<C>): (a: A) => M<C>

}

function bind<A, B, M>(m: Monad<M>, ma: M<A>, f: (a: A) => M<B>): M<B> {

return m.compose(() => ma, f);

}

방금 정의한 bind 함수는 아래와 같이 사용할 수 있습니다.

declare const IOMonad: Monad<IO>;

declare function print(message: string): IO<void>;

declare function get(message: string, defaultValue: string): IO<string>;

declare function unsafePerformIO<A>(ioA: IO<A>): A;

const main = bind(

IOMonad,

print('Welcome to monadic world'),

\_ => bind(

IOMonad,

get('What\'s your name?', 'anonymous user'),

username => print(`Hi, ${username}`)

),

);

unsafePerformIO(main);

이 코드에서 주목할 만한 점은, unsafePerformIO 함수의 호출 이전까지는 모두 순수한 연산이라는 점입니다. 입출력 연산의 동작은 순수하지 않으나, 입출력 연산 그 자체는 순수하기 때문에 입출력 연산을 조립해나가다 마지막에 실행시키는 방법으로 그 전까지의 모든 연산은 순수하게 유지한 것이지요. 그리고 이 기발한 방식의 중심에는 입출력 연산의 합성 가능성을 보장해주는 모나드가 있습니다!

***Notice****: 작동하는 실제 코드는*[*여기*](https://www.typescriptlang.org/play/?ssl=65&ssc=23&pln=1&pc=1#code/PQKhAIFgChwhJA5ANwKbgK4GdUBNwAuATgJYDGA1oQPbioB2Z1uJ9A5uABIDSAKgBTUi4AGYkiWAgFoyAGwCGWLODYNUpMlgCUMOOACa1DODLz6JgBapK4ALYBPQuJutCFksrYeCALnAWCAgAHLB9gYC9JADovAgsMACMopltgAFEAOWRxanpbBgIAZVQCYABGADYKgCZ5MoAWAAYRAE4WgGYADjKEsk6AVhFcCoTUfrLrXDIEhP7dCBBgGBhUAA8goQJwVgJ1EXkydB5ecABvXXAg+SJ5Wz8Megp6agB3egBuC6JULAxZX0wj2eb0+0AAvstoGsNkQtgR7EF0ABBIJBWT2AA8Ijoq129FwymOABpwKsAHwXAC84H42IAZGdLtdbn5VuAwVoANqIb6-f6IAC6oJgOz2B3QAFlcvJcBiJTi8QSuHwyYyLkEMN8MUiyfx5H4kVoDaj0XKSTrQeATNRbBscNqSQAhEkAYV1Ij8eoNWnAlNVKLRmIlTrJJLYnv14EdPr94ADpuD4DdRppkcNvv9JqDrrJoIh0Bg8MR4AASqh5LIAOpCWT4an0VBoIig0VEfaHcDwADyCoYSuOZy+Pz+AP43wr1aItb8ZYnNdwMdVnLiHm5VxutkFJNnVfnQpg+ZgIgeZAIJFygKw8hEqAACnshLZu9rdeekcbAxju+aySmkYPYGtehJHAcddynXAZ3LcDa19QFcFQMQG3wRRS2gyda2FQDvgITVzDfMd0PnLlGn3cFIWPRgzwvBJWFlJEnRJeU1kVQkVX4O5wClegZTlUNtmod84yzM0434j1U29DNhM-RNHV-D8ExDACrRwvC7GSG07VQfgAH1pLfEkRC0fgHgQpC8C0PMKJPajzCCUh6AIDifivVQ-EkRy2D-ETv3AZBqBIXBVXObCSnUsCMLrVV+CYYDqFkVAolkag2BcpR5FULQSU5MzENYPASUivcrIPGyqPPcxVGc-IMvc8BPNYDhqUQRASXM+QRwANQrDBUA84gmrg1qfM-PzGvYEKh1wohzGKiDpM5BytJq1zMtQH0AH5NvADrut61AiqIiCyMPaA4pA7tuJlPxrtlbtVWpUKrQ1LUdS9ONRtNPydRUq1QPC2aAbnBbY05eQjpB2syKtMEiQuFJtIdKMc1pCMpNjeNMT8+Sw3RqNFxk76uxzFMPvTTHfJJpMpsA1TAfMeRpPm2DY2e-6rQurYlu+ZAyz5AgSQcxsdyigU4JEPUTJZhcsI5+mZqq-hhb54d-hMlXReIy1-vzWGyoLc7chA2x5FcalaPxfgLiu6VcHhwCHJ2fhEErVBZBSdACFoWw7fIcAXnnRBsoufTY0t3Brbpzsuzuh2Oeql3KwseQCAAHUQZR7CMYQePyTa2vARAzFyexfewTAcCIYP4-+7B1Dz9BYydpz+AAA04EgSQAElOevZtuVAwTbnRAJD6BSsNh4rxve820fbsOLN+grKAA)*에서 보실 수 있으십니다.*

이렇게 두 관점에서 관찰한 내용들을 합쳐 보았습니다. 이제 모나드가 무엇인지 정의를 내릴 때가 되었군요. 첫 번째 관점에서 살펴본 **모나드의 정의는, 모나드인 타입이 나타내는 것이 합성 가능한 연산임을 보장하기 위한 요소들로 구성**되어 있었습니다. 또한, 두 번째 관점에서 살펴본 **모나드의 의의는, 합성할 수 있는 연산을 정의하고 추상화 할 수 있다**는 데에 있었습니다. 이를 종합하여 모나드를 정의하자면:

**모나드란, 합성할 수 있는 연산입니다.**

**마무리**

고생 많으셨습니다. 지금까지 여러분은 모나드의 거의 모든 것을 배우셨습니다. 남은 건 제가 그려드린 밑그림을 구체적인 지식들로 채워나가실 일 뿐입니다. 물론 앞으로의 일들이 마냥 쉽지만은 않을 겁니다. 그동안 이해한 것이 부정당할 때도 있고, 머릿속이 혼란스러워 질 때도 있으실 겁니다. 그렇지만 제가 이것 하나만은 보장해 드리겠습니다. 여기까지 잘 인내하면서 공부하신 여러분이라면 언젠가는, 머지않아 모나드를 꼭 이해하게 되실겁니다. 그러니 절대 낙담하지 마시고, 계속해서 나아가세요.

그럼, 마지막으로 오늘 다룬 내용을 정리하며 글을 마치도록 하겠습니다. 긴 글 읽어주셔서 감사합니다.

* 모나드의 겉뜻: **어떤 타입 M에 대해 두 함수, pure과 compose가 존재할 때, M은 모나드이다.**
* 모나드의 속뜻: **어떤 것이 모나드임이 시사하는 바는 그것이 합성될 수 있는 연산이라는 점이다.**
* 모나드: **모나드란, 합성할 수 있는 연산이다.**

**읽을거리**

* [Category theory for programmers](https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/)  
  모나드의 고향인 카테고리 이론을 프로그래밍의 개념들을 통해 설명하는 시리즈입니다. 모나드 이외에도 정말 많은 카테고리 이론의 개념들이 설명되어 있습니다. 카테고리 이론을 정석대로 이해하고 싶으시다면 이 시리즈를 추천드립니다. 덧붙이자면, 유튜브에 실제 강의를 찍은 [영상](https://www.youtube.com/watch?v=I8LbkfSSR58&list=PLbgaMIhjbmEnaH_LTkxLI7FMa2HsnawM_)도 있습니다. 영상을 선호하신다면 유튜브에서 보시는 걸 추천드립니다.
* [Monads are just monoids in the category of endofunctors](https://blog.merovius.de/2018/01/08/monads-are-just-monoids.html)  
  모나드가 수학적으로 무엇인지 빠르게, 그러나 너무 얕지는 않게 익혀보고 싶으시다면 이 글을 추천드립니다. 선수 지식들을 포함한 속성 강좌입니다.
* [A categorical view of computational effects](http://www.math.jhu.edu/~eriehl/lambda.pdf)  
  프로그래밍에서 모나드를 도입한 이론적인 배경을 설명하는 슬라이드입니다. 실제 컨퍼런스 발표의 슬라이드로, 유튜브에서 [영상](https://www.youtube.com/watch?v=Ssx2_JKpB3U&t=2036s)을 보실 수도 있으니 영상을 선호하신다면 유튜브에서 보시는 걸 추천드립니다.
* [Monads for functional programming](https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf)  
  프로그래밍의 관점에서 모나드를 설명하는 글입니다.
* [Computational lambda-calculus and monads](https://person.dibris.unige.it/moggi-eugenio/ftp/lics89.pdf)  
  모나드를 통해 연산을 표현하는 방법이 처음으로 소개된 유지니오 모기 박사님의 논문입니다. 모나드의 시작이 궁금하시다면 보시는 걸 추천드립니다.
* [Comprehending Monads](https://ncatlab.org/nlab/files/WadlerMonads.pdf)  
  처음으로 실용적인 측면에서 프로그래밍에 모나드를 도입한 필립 와들러 박사님의 논문입니다. 실용적인 모나드의 시작이 궁금하시다면 보시는 걸 추천드립니다.
* **이 글의 출처** [**https://overcurried.com/3%EB%B6%84%20%EB%AA%A8%EB%82%98%EB%93%9C/**](https://overcurried.com/3%EB%B6%84%20%EB%AA%A8%EB%82%98%EB%93%9C/)