Chapter 1

Clean Slate

프로그래밍은 타입(types)과 함수(functions)로 시작됩니다. 아마도 타입과 함수에 대해 어떤 선입견을 가지고 있을 것입니다: 그것들을 버리세요! 그것들은 당신의 마음을 혼란스럽게 할 것입니다.

하드웨어가 어떻게 구현되는지에 대해 생각하지 마세요. 컴퓨터가 무엇인지는 단지 여러 계산 모델들 중 하나입니다. 그것에 집착해서는 안 됩니다. 당신의 마음 속에서 계산을 수행하거나, 펜과 종이를 사용하여 계산을 할 수 있습니다. 물리적 기초는 프로그래밍의 아이디어에는 중요하지 않습니다.

1.1 Types and Functions

노자[1]: *서술될 수 있는 유형은 영원한 유형이 아닙니다*. 다시 말해, 유형(type)은 원시적 개념입니다. 이는 정의될 수 없습니다.

대신 그것을 *타입*이라고 부르는 대신, 우리는 그것을 *객체* 또는 *명제*라고 부를 수도 있습니다. 이는 수학의 다른 분야 (타입 이론, 범주 이론, 논리)에서 그것을 설명하는 데 사용되는 용어들입니다.

유형(type)은 하나 이상 있을 수 있어서 그들을 명명할 방법이 필요합니다. 우리는 손가락으로 가리켜서 할 수 있지만, 다른 사람과 효과적으로 소통하기 위해서 우리는 보통 그들을 이름 짓습니다. 그래서 우리는 유형 a, b, c; 또는 `Int`, `Bool`, `Double` 등을 이야기할 것입니다. 이것들은 단지 이름일 뿐입니다.

타입 자체로는 의미가 없습니다. 그것을 특별하게 만드는 것은 다른 타입과 연결되는 방식입니다. 이러한 연결은 화살표로 설명됩니다. 화살표는 하나의 타입을 소스로, 다른 하나의 타입을 타겟으로 가집니다. 타겟이 소스와 동일할 수도 있으며, 이 경우 화살표는 루프를 형성합니다.

타입 간의 화살표는 *함수*라 부릅니다. 객체 간의 화살표는 *사상*(morphism)라 부릅니다. 명제 간의 화살표는 *함의*(entailment)라 부릅니다. 이들은 단지 수학의 다른 영역에서 화살표를 설명하는 데 사용되는 단어들입니다. 이 용어들은 번갈아 사용할 수 있습니다.

명제는 참일 수 있는 것입니다. 논리에서, 우리는 두 객체 사이의 화살표를 a가 b를 수반한다(a entails b) 또는 b가 a로부터 도출 가능하다(derivable from a)고 해석합니다.

[1] 현대 표기법으로는 Laozi이지만, 전통적인 표기를 사용할 것입니다. 노자는 도덕경(또는 Daodejing)의 반전설적인 저자입니다.

두 타입 사이에 하나 이상의 화살표가 있을 수 있어서, 우리는 그것들에게 이름을 붙여야 합니다. 예를 들어, 여기 타입 a에서 타입 b로 가는 f라는 이름의 화살표가 있습니다.

$$a \xrightarrow{f} b$$

이를 해석하는 한 가지 방법은 함수 f가 타입 a의 인자를 받아 타입 b의 결과를 생성한다고 말하는 것입니다. 또는 f가 a가 참이라면 b도 참임을 증명하는 것이라고 말할 수도 있습니다.

참고: 타입 이론(type theory), 람다 계산법(lambda calculus, 프로그래밍의 기초), 논리(logic), 카테고리 이론(category theory) 사이의 연결은 Curry-Howard-Lambek 대응(Curry-Howard-Lambek correspondence)으로 알려져 있습니다.

1.2 Yin and Yang

객체는 그것의 연결로 정의됩니다. 화살표는 두 객체가 연결되었다는 사실을 증명하는 증거입니다. 때때로 증거가 없을 때, 객체들은 연결되지 않습니다. 때때로 많은 증거들이 존재합니다. 그리고 때때로 단 하나의 증거—두 객체 사이의 유일한 화살표(arrow)만이 존재합니다.

무엇이 *유일한(unique)* 것을 의미할까요? 이는 당신이 두 개의 그것을 찾을 수 있다면, 그것들은 같아야 함을 의미합니다.

모든 객체로 나가는 유일한 화살표를 가지는 객체는 *초기 객체(initial object)*라 불립니다.

그 쌍대는 모든 객체로부터 유일한 들어오는 화살표를 가진 객체입니다. 이것을 *끝 객체*라고 합니다.

수학에서 초기 대상(initial object)은 종종 0으로 표시되고, 종말 대상(terminal object)은 1로 표시됩니다.

0에서 임의의 객체 a로 가는 화살표는 $¡_a$로 표시되며, 종종 ¡로 축약됩니다.

어떤 객체 a에서 1로 가는 화살표는 $!_a$로 표시되며, 종종 !로 줄여서 사용됩니다.

초기 객체(initial object)는 모든 것의 근원이다. 타입(type)으로서 이는 Haskell에서는 `Void`로 알려져 있다. 이는 모든 것이 발생하는 혼돈을 상징한다. `Void`에서 모든 것으로 향하는 화살표가 존재하므로, `Void` 자신에게로 향하는 화살표도 존재한다.

그러므로 `Void`는 `Void`와 다른 모든 것을 낳습니다.

끝 객체(Terminal object)는 모든 것을 통합합니다. 타입으로서 이는 유닛(Unit)으로 알려져 있습니다. 이는 궁극적인 질서를 상징합니다.

논리에서, 종말 객체(terminal object)는 궁극적인 진리를 의미하며, T 또는 `T`로 기호화됩니다. 어떤 객체로부터 이것으로의 화살표(arrow)가 존재한다는 사실은 `T`이 당신의 가정이 무엇이든 상관없이 참이라는 것을 의미합니다.

반대로 초기 객체(initial object)는 논리적 거짓(falsehood), 모순(contradiction) 또는 반사실(counterfactual)을 의미합니다. 이는 False로 작성되며 거꾸로 된 T, ⊥로 기호화됩니다. 그것으로부터 모든 객체로 화살표가 있다는 사실은 거짓 전제에서 시작하여 무엇이든 증명할 수 있음을 의미합니다.

영어에는 반사실적 함축(counterfactual implications)을 위한 특별한 문법 구조가 있습니다. 우리가 "만약 소원이 말이라면 거지들이 탈 것이다."라고 말할 때, 우리는 소원과 말의 같음을 거지들이 탈 수 있게 한다는 의미를 내포합니다. 그러나 우리는 그 전제가 거짓임을 알고 있습니다.

프로그래밍 언어는 우리가 서로, 그리고 컴퓨터와 소통할 수 있게 해 줍니다. 어떤 언어들은 컴퓨터가 이해하기 쉽게 되어 있고, 다른 언어들은 이론에 더 가깝습니다. 우리는 타협점으로서 Haskell을 사용할 것입니다.

Haskell에서, 터미널 타입의 이름은 `()`, 빈 괄호 쌍으로, 이를 유닛(Unit)이라고 발음합니다. 이 표기법은 나중에 더 이해가 될 것입니다.

Haskell에는 무한히 많은 유형이 있으며, 각 유형에는 `Void`로부터의 고유한 함수/화살표가 있습니다. 이 모든 함수는 동일한 이름으로 알려져 있습니다: `absurd`.

Programming	Category theory	Logic
type	object	proposition
function	morphism (arrow)	implication
`Void`	initial object, 0	False $\bot$
`()`	terminal object, 1	True $\top$

1.3 Elements

객체는 부분을 가지지 않지만 구조를 가질 수 있습니다. 구조는 객체를 가리키는 화살표들에 의해 정의됩니다. 우리는 화살표를 통해 객체를 *탐사*할 수 있습니다.

프로그래밍과 논리에서는 초기 객체가 구조를 가지지 않기를 원합니다. 그래서 우리는 초기 객체가 되돌아오는 화살표(자기 자신으로부터 되돌아오는 화살표를 제외하고)는 가지지 않는다고 가정할 것입니다. 따라서 `Void`는 구조를 가지지 않습니다.

종점은 가장 단순한 구조를 가집니다. 어떤 객체로부터 이것으로 들어오는 화살표는 하나만 있습니다: 이것을 어떤 방향에서든 조사하는 방법은 하나뿐입니다. 이 점에서, 종점은 하나의 나누어지지 않는 점처럼 행동합니다. 그것의 유일한 속성은 그것이 존재한다는 것이며, 다른 객체로부터 오는 화살표가 그것을 증명합니다.

터미널 객체는 매우 단순하기 때문에 이를 사용하여 더 복잡한 다른 객체들을 탐구할 수 있습니다.

만약 어떤 대상 a로부터 종료 객체(terminal object)로 가는 화살표(arrow)가 하나 이상 있다면, 그것은 a가 어떤 구조를 가지고 있다는 것을 의미합니다: 그것을 바라보는 방법이 여러 가지임을 의미합니다. 종료 객체가 마치 점처럼 행동하기 때문에, 우리는 종료 객체로부터 나오는 각 화살표가 그것의 목표(target)의 다른 점(point) 또는 요소(element)를 선택하는 것으로 시각화할 수 있습니다.

범주 이론에서 x가 a의 *글로벌 요소*(global element)라면 그것은 화살표(arrow)입니다.

$$1 \xrightarrow{x} a$$

우리는 종종 그것을 단순히 요소(어떤 경우엔 "전역"을 생략하고)라 부릅니다.

유형 이론에서, $x : A$는 x가 유형 A에 속한다는 것을 의미합니다.

Haskell에서는 대신 더블 콜론 표기법을 사용합니다:

```
x :: A
```

Haskell은 구체적인 타입(concrete type)에는 대문자 이름을 사용하고, 타입 변수(type variable)에는 소문자 이름을 사용합니다.

우리는 `x`가 `A` 타입의 항(term)이라고 말하지만, 범주론적으로는 이를 화살표 $x : 1 \to A$, 즉 `A`의 전역 원소(global element)로 해석합니다. [2]

[2] Haskell 타입 시스템은 `x :: A`와 `x :: () -> A`를 구분합니다. 그러나 범주론 의미에서는 동일한 것을 나타냅니다.

논리에서, 이러한 x 는 A 의 증명(proof)이라고 불리니, 이는 함의 $\top \to A$ (만약 **참**(True)이 참이면 **A** 도 참이다) 에 대응하기 때문입니다. A 의 증명은 여러 개가 있을 수 있음을 주목하십시오.

우리가 `Void`로 향하는 다른 객체로부터의 화살표가 없다고 규정했기 때문에, 최종 객체에서 그것으로 향하는 화살표도 없습니다. 따라서 `Void`는 요소가 없습니다. 이것이 `Void`를 비어있다고 생각하는 이유입니다.

종말 대상은 하나의 원소만을 가지는데, 이는 자신으로부터 자신으로 오는 유일한 화살표가 있기 때문입니다, $1 \to 1$. 이것이 우리가 때때로 이를 단일집합(singleton)이라 부르는 이유입니다.

참고: 범주 이론에서는 초기 객체가 다른 객체로부터 오는 화살표를 가지는 것에 대한 금지가 없습니다. 그러나 여기서 우리가 공부하고 있는 데카르트 닫힌 범주(카테고리)에서는 이것이 허용되지 않습니다.

1.4 The Object of Arrows

화살표(Arrows)는 두 객체 사이에 집합[3]을 형성합니다. 이것이 카테고리 이론을 공부하기 위해 집합 이론에 대한 지식이 필요한 이유입니다.

프로그래밍에서 우리는 `a`에서 `b`로 가는 함수들의 *유형*(type)에 대해 이야기합니다. Haskell에서는 다음과 같이 씁니다:

```
f :: a -> b
```

의미는 `f`가 "`a`에서 `b`로의 함수" 타입을 갖는다는 것입니다. 여기서 `a->b`는 우리가 이 타입에 부여하는 이름입니다.

함수형 타입들을 다른 타입들과 동일하게 취급하고 싶다면, `a`에서 `b`로의 화살표 집합을 나타내는 *객체(object)*가 필요합니다.

이 객체(object)를 완전히 정의하기 위해서는, 우리가 해야 할 일은 특히 `a`와 `b`와의 관계를 기술하는 것입니다. 우리는 이를 수행할 도구들이 아직 없지만, 곧 그 지점에 도달할 것입니다.

현재로서는 다음과 같은 구분을 기억해 두시면 됩니다: 한편으로는 두 개의 객체 `a`와 `b`를 연결하는 화살표(arrow)들이 있습니다. 이 화살표들은 하나의 집합을 형성합니다. 다른 한편으로는 `a`에서 `b`로 가는 *화살표의 객체*(object of arrows)가 있습니다. 이 객체의 "원소"(element)는 최종 객체 `()`에서 우리가 `a->b`라고 부르는 객체로 가는 화살표로 정의됩니다.

우리가 프로그래밍에서 사용하는 표기법은 이러한 구분을 흐리게 하는 경향이 있습니다. 이러한 이유로 범주 이론에서는 화살표의 객체(object of arrows)를 *지수적*이라 부르고 그것을 b^a 로 표기합니다 (원천 객체(source object)가 지수에 있습니다). 그래서 다음과 같은 문장을 쓸 수 있습니다:

```
f :: a -> b
```

와 동등합니다.

$$1 \xrightarrow{f} b^a$$

논리학에서 화살표 $A \to B$ 는 함의를 나타냅니다: 이는 "만약 A라면 B이다"라는 사실을 진술합니다. 지수 개체(exponential object) B^A 는 대응하는 명제입니다. 그것은 참일 수도 있고 거짓일 수도 있습니다, 우리는 알 수 없습니다. 당신은 그것을 증명해야 합니다. 그러한 증명은 B^A 의 한 요소입니다.

B^A의 원소를 보여주면 B가 A로부터 따라 나온다는 것을 알 수 있습니다.

[3] 엄밀히 말하면, 이는 *국소적으로 작은* 카테고리에서만 성립합니다.

다시 한 번 “소원이 말이라면 거지는 말을 탈 것이다”라는 문장을 객체로 생각해 봅시다. 이 문장은 비어있는 객체는 아닙니다. 왜냐하면 이를 입증할 수 있는 증거를 제시할 수 있기 때문입니다. 예를 들어, “말을 가진 사람은 그 말을 탄다. 거지들은 소원을 가진다. 소원이 말이기 때문에, 거지들은 말을 가진다. 따라서 거지들은 말을 탄다.”라는 내용입니다. 그러나, 비록 이 문장의 증거를 가지고 있다고 하더라도, 이것은 아무런 쓸모가 없습니다. 왜냐하면 이 전제 “소원 = 말”을 결코 증명할 수 없기 때문입니다.