수학 언어와 인간의 사고방식, 그리고 새로운 수학적 프레임워크의 제안

20715 이인환 (수열 관련 발표)

https://faraway6834.github.io/unbeauty/privateNote/PersonalPersoanlVeryPersonal/FuckingPersonal

PART1. 수에 대한 고찰

수학과 언어

수, 수사 등은 크기 개념이다. 이러한 크기 개념은 곧 설명할 일개 사고방식에 불과하며, 이러한 수에대한 설명의 형식적 방언이 수학이다.

그러나 언어는 근본적인것들이 아니다.

시작해보자.

언어에 대한 해부를.

수란 무엇인가

이야기 하기 앞서, 말하고자 한다.

나는 이 주제에 대하여, 수는 수학적으로 정의되는 추상적 대상이며, 형이상학적인 허상 취급을 **했었다**

아래는 그 글이다

취급했던 글 : 주장

- 전제
 - 전제에 따른 함의
 - 1. 픽토그램(pictgram)에서 보존 개념(conservation)이 불필요한 이유
 - 2. 수라는 수학에서 추상적 보존 개념 (conservation)의 핵심적인 부분이 왜 형식적 논리인 수학으로 작성되어야 하는가
  - 결론

전제

우리는 외부세상을 감각으로밖에 인지할수밖에 없기에, 내부 세상(본인의 생각의 언어 L을 가정하고 L이 작동하는 본인)은 와부세상에 대해 귀납적인 추론밖에 할 수 없다.

전제에 따른 함의

그런 외부 새상에서 일관됨을 연역적으로 보일 방법은 존재하지 않으며, 따라서, 외부 새상이 형식적인 논리로 서술되는지도 연역적으로 합리화할 방법이 없다.

1. 픽토크램(pictgram)에서 보존 개념(conservation)이 불필요한 이유

다음 : 2. 수라는 수학에서 추상적 보존 개념 (conservation)의 핵심적인 부분이 왜 형식적 논리인 수학으로 작성되어야 하는가

목차

Defination of Unary

표기 Unary ≡ [UnarySystem := λF. [Char := λx. "x is charactor"][String := λx. x ∈ {c | Char(c)}□ [t := |x|]][F := λn:ℕ₀.λx:Char. ∀y (단. y ∈ {S}ⁿ)][x ← y := z (단. z → y = x)][x↓□ y := z (단. z ↑□ y = x)][ℙ₁ := ⩟ ∀String(x), "x is string" F□ ≡ F(n) x▪y ≡ \stackrel{x}{y} 「x」≡▪x 〖x〗₁ ≡ x 〖x〗₍□₊₁₎ ≡ x 〖x〗□ ▲「F□(x)」 ≡ F(n⁺)(x) ▼「F□(x))」≡ F(n⁻)(x) F□(x) 〖「▲」〗₁「F□(x)」 ≡ F₍□₊□₎(x) F□(x) 〖「▼」〗₁「F□(x)」 ≡ F₍□₋□₎(x) F□(x) 〖「▲」〗₂「F□(x)」 ≡ F₍□□₎(x) F□(x) 〖「▼」〗₂「F□(x)」 ≡ F(n÷m)(x) F□(x) 〖「▲」〗₃「F□(x)」 ≡ F(n□)(x) F□(x) 〖「▼」〗₃「F□(x)」 ≡ F(□√n)(x) F□(x) 〖「▲」〗₄「F□(x)」 ≡ F(□n)(x) F□(x) 〖「▼」〗₄「F□(x)」 ≡ F(super-root□(n))(x) F□(x) 〖「▲」〗₍₂₊□₎「F□(x)」 ≡ F(n ↑□ m)(x) F□(x) 〖「▼」〗₍₂₊□₎「F□(x)」 ≡ F(n ↓□ m)(x) x ▲ y ≡ y 〖「▲」〗₁「x」 x ▼ y ≡ y 〖「▼」〗₁「x」 x ▶ y ≡ y 〖「▲」〗₂「x」 x ◀ y ≡ y 〖「▼」〗₂「x」 F□(x) ↑□ F□(x) ≡ F□(x) 〖「▲」〗₍₂₊□₎「F□(x)」 F□(x) ↓□ F□(x) ≡ F□(x) 〖「▼」〗₍₂₊□₎「F□(x)」 F□(x) → F□(x) ≡ F(n → m)(x) F□(x) ← F□(x) ≡ F(n ← m)(x) ⩟]]

사용법

> `(∃UnarySystem(F) s.t. ⊢ ℙ₁)( ⊢ T)`

ℙ₁은 "†문자열은 문자의 튜플이고, ‡아래 구문론적 등호가 성립한다"는 뜻이기에, ℙ₁이 참일때, †문자열을 문자의 튜플로 정의하고, ‡제시한 구문론적 등호를 받아들이며, ℙ₁이 거짓일때, †문자열을 문자의 튜플로 정의하지 않고, ‡제시한 구문론적 등호를 받아들이지 않는것이다.

즉 문자열의 정의와 표기법을 허용하면 참, 그러지 아니하면 거짓이지. 따라서, `s.t. ⊢ ℙ₁`이라고 하면, F에 대해, ℙ₁이 참이라고 강제하는 뜻인것이다.

상수 "PlasticArrow" 정의 : $\mathbb{R}^2$나 $\mathbb{R}^3$범위에서 다음을 정의한다

Plastic Arrow

- `□`는 벡터공간 `𝕍 = ℕ₀¹`에서의 "기저벡터"

참고로 Unary에서 표현하는 수는 `□`가 선형생성하늑 벡터공간의 스칼라이다.

Padic Plastic Arrow

- `➪`는 초실수체 벡터공간 `𝕍 = ℚ¹`에서의 "기저벡터"

Standard Plastic Arrow

- `➡️`는 벡터공간 `𝕍 = ℝ¹`에서의 "기저벡터"

번외 : 실현?

물론 현실에 저런것들을 만들려면 「플라스틱-화살표 교구」에 "단면"이 존재해야하는게 함정이다.

그러나 "플라스틱 화살표 교구의 단면"은 "폐곡선의 방정식" `F(x, y) = 0`으로 표현하면 된다. (따라서, "플라스틱 화살표 교구의 단면의 폐곡선의 방정식 음함수 표현 F"가 존재한다.)

"실현" ㅋㅋㅋㅋ... 개연성도 없거니나와, 웃긴다. ㅋㅋ

새상은 차갑다.

시각 감각 언어 `ㄴ`을 가정하자.

정의

L은 pictogram 언어이다.

  단순히 pictogram을 나열한다.

센다는 것(算)은 pictogram을 새는것으로 정의된다.

수를 세는 함자 "산자 System"를 정의하겠다.

> `算子システム ≡ [算 := λx. RF(x)⁻¹ [R := λf.λx.λy.fyx]] (단. (∃UnarySystem(F) s.t. ⊢ ℙ₁)( ⊢ T))`

그러면,

`算(□) (算子システム)`는 **변항 □에 들어갈 pictogram을 새는 함자이다.**

> ex) `算("□")("□□□□□") = 5 (算子システム)`

이때 저 pictogram문장은, 수학언어에서 접근한것이기에, 그냥 문자열 해석이다.

문법 번역식 교수법마냥, 픽토그램을 픽토그램이 아닌 문자열로 해석했기 때문에,

이는 쐐기문으로 쓰여있는 고대 메소포타미아 문명의 언어 글자를, 숫자로 취급하는것과 같은셈이다.

따라서, pictogram과는 무관하게, 수학이 이것에 의미를 부여한 것이다.

사실 pictogram외에도 걍 문자열이면 다 셈 함수의 정의역이므로, 셈은 우리로 하여금 새상이 수로 해석될수 있다고 귀납적 추론에만 근거한 주장을 펼치게 한다.

보존 개념 (conservation)은 귀납적으로 증명되었지만, 연역적으로는 그 자체를 가정할때, 형식적 논리인 수학을 요구하기에, "증명되지 못한 (말할수 없는)것" 이며, 전조작기가 지나고, 구체적 조작기에 들어서 후천적으로 얻는, 보편적인 개념이다.

conservation을 놓고 생각해보자.

> `🍑🍍🍍🍊🍓`라는 pictgram이 있다.

해당 픽토그램이

> `🍑🍍🍊🍓`

로 바꿔도 무방하다

따라서, pictgram은 conservation을 요구하지 않는다.

따라서, pictogram새상에서도 conservation을 합리화할 방법은 존재하지 않는다.

따라서, 외부 새상이 pictogram으로 인식되어도, conservation을 합리화할 방법은 존재하지 않는다.

수라는 보편적 개념이 얼마나 보편적인 지성적인 개념일까

수에 대한 이해(「"수학적 관점에서, 외부 세상의 대상의 양이 셈으로 해석된다"는 결과를, "셈이 근본이다"라는 확대해석 • 합리화」한것이 수이다)를 보면, 수는 언어적으로 만들어진 요소이므로, 이해하지 못할수도 있다.

이러한 측면에서, 난산증은 엄청나게 보편적인 언어적 가정인 수를 이해하지 못하는 장애라고 볼수 있겠다.

Charactor/Text(String)은 Language의 Symbolization(심볼화)이며,

Symbol(기호)은 특정한 의미를 가지도록 약속된 도형(**圖形**)이다

그리고 이것은 정말로 엄청나게 보편적인 개념인데,

이러한 측면에서, 난독증은 엄청나게 보편적인 인간이 창조한 개념인 문자를 이해하지 못하는 장애라고 볼수 있겠다.

이어서,

마지막으로, 자폐증의 경우 템플 그랜딘이 말했듯, 자폐증 내부의 언어는 그림일수도 있다.

작성자 본인도 자폐증이긴 하다 (나는 내가 자폐증이여서 자폐를 싫어하긴 하니 자폐를 변호하는게 아니지만) 자폐증의 경우, 언어를 본인 맘대로 해석하는 경향을 띄는 사람이 있다 (예시 : 본인)

언어는 그 본질이 기존의 뜻으로 결정되지 않고, 필연적으로 의외성을 수반해여만 한다.

즉 일상 언어의 본질은 인간 사고 기저에 있는 의외성과 통념(정확히는 개념에 가깝다. 예를들어, 눈을 찡그린 사람은 화난것)에 있다.

이러한 의외성은,

> 인간이 보편적으로 가지는 태생적인 지능에 따른, 언어의 지능적 의외성과

> 인간이 기본적으로 가지는 통념(개념; 예를 들어 "뿌린대로 거둔다"는 하는데로 돌아온다는 뜻이라는거, 그 의외성의 핵심 개념)에 따른 언어의 통념적 의외성이 있을것이다.

따라서, 인간이 사고의 근본인 언어에 있어서, 본질적인 부분에 대한 엄청난 이해 결함을 가진 장애인으로, 일부 자폐성 장애인 (예시 : 초등학생때 본인)이 있을것이다.

이러한 관점에서, 난산증에게 결핍된 능력은, 보편적인 수에 대한 개념이라고 볼 수 있다.

이러한 수 개념은 수에 대한 어느정도의 이해를 필요로 하므로,

수에 대한 보편적인 지능과 이해능력은, 난산증을 제외한 인간이 매우 발달한것일 뿐, 본질적인것이 아니라고 볼수 있다.

정상인은 에초애 많은 부분에서 선천적인 지능이 있다.

그걸 없는걸 장애라고 하고, 솔찍히 그런 당연한거를 모르니까 약간 역겨운 감도 있다.

과학을 가져오는것에 대한 반론

과학(자연과학, 사화과학, 형식과학 등)과 같이 귀납적으로 증명된 대상들은 지금 이 연역적인 논의에서, 귀납적이지만 연역적이지 않다고 반박하니, 과학을 예시로 반박하면 안될것이다.

그렇게 되면 새상이 과학적으로 이루어졌다는 가정으로 수학을 증명하고 다시 과학을 증명하는 순환 논증에 빠질것이고, 이러한 방식으로, 새상이 과학으로 이루어졌다는 전제로, 과학의 탐구 방식이나 수학을 합리화 하는 즉시, 그것은 수학에서 금한 자기참조를 범하는것이다 (따라서, bootstraping같은 과정은, 자기참조를 범하는 합리화이므로, 연역논증이랑은 거리가 멀다 (inference to the best explanation라고 하는것도 이해가 안된다))

또한 과학은 수학을 수반하지 않아야 될 수도 있다. 만약 과학에서 수학을 쓸수 없다는게 증명됐다고 가정하자, 그러면 과학은 수학을 사용해야 한다는 가정이 깨지므로, 페러다임 시프트가 일어날 것이다.

과학이 수학을 사용해야한다는 가정은, 과학이 형식논리를 이용해야한다는 가정과 동등한 뜻이다.

그런데 과학이 형식논리를 이용해야한다는 가정은, 해당 과학이 일관된 형식적 논리로 서술될수 있다는 가정이므로, 해당 가정이 성립하기 위해서는, 논리를 사용하기 위한 규칙 (공리계, 언어, 표기 등)과 과학이 일관된 대상을 탐구해야한다는 가정을 가지게 되므로, 공리계를 사용할수 있다는 근거와 탐구하는 대상이 과학적이라는 근거가 있어야만, 연역적으로 뒷받침될수 있다.

2. 수라는 수학에서 추상적 보존 개념 (conservation)의 핵심적인 부분이 왜 형식적 논리인 수학으로 작성되어야 하는가 : 두번째 글 : 논리적으로 다룬다 전재할때, 대수식은 논리적으로 그 뜻이 해석 • 계산된다.

이 내용은 PPT 후반 63슬라이드 에서 다루겠다.

결론은, 수학에서 말하는 "추상적 보존 개념 (conservation)의 핵심으로써의 수"는 수학 없이 독자적으로 성립할수 없다. (단. 언어에서 수사같은 거나 숫자같은거는 수학적으로 해석하는것이 아니기에 본인은 수로 취급하지 않았다. 언어에서는 수에 의외성을 가지고 해석할수 있기에, **실제로 일관되지 않는다**; 시에서 갑자기 하나가 두개가 될수 있다.)

결론

따라서, 픽토그램으로 우리가 새상을 받아들인다 쳐도, 픽토그램 새상에서도 수, 수학, 보존개념은 요구되지 않으며, 해당 개념들은 언어를 통한 부가적인 개념일 뿐이지, 비물질적 실체가 아니다.

이 글은 틀렸다

-

해당 픽토그램을 해석함에 있어서, 우리는 기저에 깔린 지식을 이용하여 그 갯수를 알수 있다.

선험적이거나 경험적이거나 언어적이거나 본능적인것 아니냐는 질문에 틀렸다고 반박했다고 볼수 없는 글이다.

Step 1. 생각의 선행

생각은 내용을 가진다, 여기서 생각은 그냥 직설적인 생각이며, 내용은 상자안의 내용물 같지만 추상적이라 설명할수 없다. 나는 그 내용이 글이라고 한적이 없다. 내용은 내용이며 그 내용이 무엇인지 우리는 말할수 없다.

생각의 다음 생각에 대해서도 단정하면 안된다. 생각의 다음 생각이 어떻게 되는건지 우린 모른다.

그러나 우리는 생각하려한다, 스스로 정신적인 방법으로 식물인간이 되고자 하지 않는다, 물론 의식적으로 "나는 정신적인 방법으로 식물인간이 될거야"라고 생각한다면 생각이다.

인간이 고등하다는 말에 속뜻, 그 생각을 버리자.

말의 속뜻중 의심되는건 일단 다 버리고 들어보라.

정의들

생각 다음 생각을 정해보는것, 생각을 전파하는것, 생각을 당연한 생각 다음 생각 방식으로, 나의 기본적인 능력으로 이해하는것, 그리고 외부에서 기인한 일로 생각이 추가되는것. 이러한 것을, "개척"/"전파"/"이용"/"추가"라고 하며, 우리 생각이 이러한 특성을 가진다는것이 공리이다.

여기에서 추가는 유일하게 능동적으로 하는게 불가능하다. (요청하는것도 수동적으로 받아들이게 해달라는거니)

그리고 전파는 생각을 공유하는거다. 전파를 받는거는 모른다.

그러나 이용하는것은, 당연한 생각 다음 생각을 이용한다. 이러한 당연한 생각 다음 생각으로 능력을 말위하는걸 잠재 능력이라 하여, 재능이라 한다.

언어와 크기

우리는 대부분, 언어에 대한 재능으로 언어를 습득할 능력을 가지고 있고, 크기를 습득할 능력을 가지고 있다.

이것들은 재능인 경우가 대부분이지만 특수한 경우를 보자면 아니다.

언어와 크기는 습득하니까, 생각이다.

언어와 크기가 전파되었으니 말이다.

크기 감각

사실 크기같은 경우에는 전파해준 사람 없이도 사물에서 기인하여 나오기도 한다.

어떻게 된지는 몰라도 이미 존재한거고, 우리는 그것우로 생각하며 우리는 그것에 대한 관점화된 틀을 벗을수 없는 부분이 있다.

이러한 벗을수 없는 "단단한 틀"이 있다. (예시 : 윤석열 하면 계엄이 떠오른다... 농담ㅋㅋ 이런건 예시로 적절치 않군 ㅋㅋ)

생각이 선행한다.

이러한 개념들이 이용을 통해서 추가되는경우가 매우 많다.

특히 살아남기 위해 필요하면 더더욱.

우리는 생각하려는 의지를 가지며, 언어와 크기는 그것의 대표격이다.

그러한 능력이 선천적인 필수 루트가 아닌 이유는, 그렇지 않은 예외경우가 존재하기 때문이다.

내가 본 대부분은 언어습득능력이나 크기습득능력을 생습적 재능으로 가진다.

그러나, 이러한것은 생각흐름을 탁! 바꿀수 있거나 그 생각흐름을 주는 경우들이 많으며,

이러한 "기저에 깔린" 생각들은 생각이지 인간의 필수 능력이 아니다.

생각이 선행한다.

Step 2. 크기 개념

크기 개념을 생각하는것은대부분 된다.

생각하지 못했다 해서 난산증을 차별할 근거가 되지 않는다는것도당연하다.

정량화된 크기 개념이 수(數)이며, 수학 언어는 수를 다루는 언어이다.

그런데, 수학 언어는 형식 언어로 설명된다.

따라서, 크기 개념이나 수학을 다룸에 있어서, 크기나 수학에 기초한 서술을 접근할수 없는 사람이 있음은 매우 당연하다.

언어 혹은 수(數)나 수학적 서술이나 크기 개념은 우리가 생각할수 있도록 익혀진 상태일때야 사용 가능한 것이지, 재능이 아닌 이상 생습적이지않다

한 문장으로 정리하겠다.

다른 관점 말고, 생각하는 관점에 있어서,

언어나 수, 수학적 서술과 크기는, 우리가 생각하는 것에 있어서 생각이며, 재능으로 타고나지 않으면 본능이 아니다.

이걸 얼마나 더 쉽게 설명해야 할지 모르겠다.

재능이라거나 이런말보다는 생존을 위한 (사자들에게도 크기 개념은 있음) 생각, 사념체일 뿐.

이상.

의외성 정리

용어 정의

- True Mean (참뜻) : 의외성(속뜻)

- Shell Mean (껍대기 뜻; 쉘 민) : 겉뜻

- 속뜻 없음 (Exceptless) : Shell Mean = True Mean

- 초완전성 (Hyper-Completeness) : "This sentence is False"를 허용하는것.

- Black and White Proposol : ⊢ (¬Hyper-Completeness)

- Simply Mean Proposol : ⊢ Exceptless

- 단순언어 : Simply Mean Proposol이 항진인 언어

흑백논리와 형식언어의 정체에 대한 고찰

Black and White Proposol를 공리로 하는 논리를 흑백논리라고 부를수 있음이 당연하다

또한 흑백논리중 단순언어인것이 형식언어인것이라도 보면 된다.

정리 문단에 따른 보조설명

우리 언어는 보이는대로 해석해야하는것과 아닌것이 있어,

보이는대로 해석해야하는것을 단순언어 (이 경우 언어 해석에 예외가 없이 보이는대로 닥치고 그뜻이다)

그리고 단순 언어가 아닌 언어 (이 경우, 언어 해석에 여러 의외성이 끼어들기에, 단순히 보이는 뜻을 뜻으로 단정할수 없다)가 있다

또한 논리는 장자철학처럼 "갓나서 죽은 아기보다 오래 산 사람은 없으니 팽조(760살이 넘게 살았다는 전설 상의 신선)도 일찍 요절한 사람이다"가 맞을수 도 있지만,

흑백논리에서는, Black and White Proposol을 참으로 하여, x이면서 동시에 x가 아닌것은 불가능하다. (Fun Fact : 흑백논리이면 단순언어임이 논리적 귀결이다)

공리

1. 말의 뜻은 True Mean과 Shell Mean이 있다

TrueMean Theroem

Hyper-Completeness 일때도 "말의 뜻 ≠ True Mean"마저 True Mean으로 True Mean(결국 전제로 한 참인 문장에서 연역(이때는 초완전땜에 가능)으로 "말의 뜻 = True Mean")이고 (Hyper-TrueMean Lemma)

Hyper-Completeness 가 아닐때도, "말의 뜻 = True Mean"이므로, 말의 뜻 = True Mean으로 (Formal-TrueMean Lemma)

말의 뜻은 True Mean을 말한다. (TrueMean Theroem)

A. Formal-TrueMean Lemma

- ¬Hyper-Completeness, ⊭ 말의 뜻 ≠ True Mean ⊢ 말의 뜻 = True Mean

B. Hyper-TrueMean Lemma

- Hyper-Completeness, 말의 뜻 ≠ True Mean ⊢ 말의 뜻 = True Mean

C. TrueMean Theroem

- 말의 뜻 = True Mean

Proof

1. 말의 뜻 ≠ True Mean [Hyp]

2. Hyper-Completeness [Hyp]

3. "말의 뜻 ≠ True Mean"라는 점도 True Mean임

[Paradoxic Lemma]

3. "말의 뜻 ≠ True Mean"라는 점도 True Mean이고 참이기에, 말의 뜻 = True Mean임

4. 말의 뜻 = True Mean

이하에서,

말의 뜻 ≠ True Mean, Hyper-Completeness ⊢ 말의 뜻 = True Mean ⋯ (1)

1. 말의 뜻 ≠ True Mean [Hyp]

2. ¬Hyper-Completeness [Hyp]

3. "말의 뜻 ≠ True Mean"라는 점도 True Mean임 [Paradoxic Lemma]

4. 모순

이하에서, ⊬ 말의 뜻 ≠ True Mean ⊢ 말의 뜻 = True Mean

(1), (2) ⊢ 말의 뜻 = True Mean

Q.E.D.

해설

항상 True Mean만 말의 뜻임을 증명하자,

True Mean이 뜻이 아닌 말이 있다고 가정하자,

그렇다면 그 말은 True Mean이 뜻이 아니라는 뜻이 True Mean이 된다

이것이 Paradoxic Lemma다

이하에서 Hyper-Completeness에 따라 참인 경우와 거짓인 경우로 나누어 논증하자

각각

상황 1. Paradoxic Lemma에서, Hyper-Completeness인 상황

Paradoxic Lemma가 참이될수 있으므로, Hyper-TrueMean Lemma가 참이다

상황 1 종료

상황 2. Paradoxic Lemma에서, 비 Hyper-Completeness인 상황

Paradoxic Lemma이 모순이므로, 전재인 "말의 뜻 ≠ True Mean"이 거짓이다.

따라서, Formal-TrueMean Lemma가 참이다

상황 2 종료

이하에서,

상황 1, 상황2에 따라, 연역,

항상 True Mean이 말의 뜻이 된다.

따라서, TrueMean Theroem이 참이다

Q.E.D.

Exceptless Thorem

Exceptless ⊢ 말의 뜻 = Shell Mean

Proof)

1. Exceptless [Hyp]
2. 말의 뜻 = True Mean [TrueMean Theroem]
3. Shell Mean = True Mean
4. 말의 뜻 = Shell Mean [결론]

이하에서, `Exceptless ⊢ 말의 뜻 = Shell Mean`임이 당연하다.

해설

앞서 증명한 TrueMean Theroem에 따라,

말의 뜻 = True Mean

Shell Mean = True Mean 이면, 그리고 이때만 Shell Mean = True Mean이다

(쉽게말해 A = B = C니 A = C)

PART 2. 근본 있는 함수 인자 기반 연산의 활용 수학 체계 제안 (방정식 유사; 마인드는 연산의 서술)

소개

나는 때때로 수학이 오만하며, 급조된것 같다는 인상을 많이느껴, 이번에 제안을 해보고자 한다.

ALKALIC

Alkalic : Alkalic Linear-algebra + Königsberg Axiom + Lambda Incoding Calculate

심지어 모델론에서도 …

M = (ℕ, 0 = ∅, s(x) = x ∪ {x}) 대신

수열의 곱을 다가함수로 써서,

M = Π<ℕ, [0 := ∅], [s := λx. x ∪ {x}]>

같은 서수의 정의가 가능하다는 점에서,

이제 Structure와 변수 대입까지 함수 안에서 됐음.

이제 구조체도 non-structial 논리적 귀결에서 씀으로 쓸수있음

Alkalic Linear-algebra

∀x (각각 유일)∃!n(x) s.t. n = ObjectID(x) ∈ Scala

- AlkalicVectorSpace = Scala□ [t := |Scala|]
- AlkalicMetrixSpace = AlkalicVectorSpace□ [t := |Scala|]
- SetTheorem ∈ AlkalicMetrixSpace
  - Notation Definition m ∈ n ≡ SetTheorem□□(□)□□(□) [oi := ObjectID]

Lambda Incoding Calculate

Alkalic Algbra서 AlkalicVectorSpace나 oidfield = Σ□ ObjectID e□에 대해, 입력받는 Tensor입력으로 람다, 대수함수, Alkalic Algbra서 다가함수를 포함한 함수 구현.

변항은 이론에 인자로 설명 가능

Königsberg Axiom

```
⊢ KönigsbergAxiom(x, y, Φ) := (x = y ↔ (Φ ↔ (Φ [x := y])))
```

TIP: 대입이에요

(오류있어서 수정할거)

Alkalic의 형식증명 ; `Alkalic-Proofmood`

`규칙 `using x = y → (Φ ↔ (Φ [x := y])))`

```Alkalic-Proofmood
□.1. using x = y → (Φ ↔ (Φ [x := y]))
□.2. x = y
□.3. Φ
□.4. Φ [x := y]
□.5. Φ ↔ (Φ [x := y])
```

규칙 `using (Φ ↔ (Φ [x := y]))) → x = y `

원리 : `x = y ↔ (Φ ↔ (Φ [x := y])))` 서 `Φ ↔ (Φ [x := y]))` 이며, 그것이 결론 (5번 라인)과 같음을 보임

```Alkalic-Proofmood
□.1. using (Φ ↔ (Φ [x := y]))) → x = y
□.2. Φ
□.3. Φ [x := y]
□.4. Φ ↔ (Φ [x := y])
□.5. x = y
```

내부적으로 결론(5번 라인)이 참일때만 계속 동작함, 또한 결론은 리스트에 쌓이면서, 마지막 줄인 Theorem에 도달할때까지 Lemma가 리스트업되서, 문장이 참인지는 Lemma로 보임.

규칙 : `Starting Listup Hyperthesis`

미리 Hyperthesis나열을 시작함

규칙 : `Quit Listup Hyperthesis`

더이상 Hyperthesis를 받지 아니함

규칙 : `Starting Another Subproof`

새 스택프레임을 만들어, 새로운 부분증명을 시작함

규칙 : `Quit Another Subproof`

부분증명을 끝내, Lemma List에 추가하고, 스택프레임을 pop함

HAPA - I

규칙 : `APAristotel-y` (nonHyperVersion 형식증명 only)

>

> HAPA Theorem이라는 외부정리를 이용하여서, y = x이고 y = z이면 x = z임을 보임

>

> 규칙 : `APAristotel-z` (nonHyperVersion 형식증명 only)

> HAPA Theorem이라는 외부정리를 이용하여서, x = z이고 y = z이면 y = x임을 보임

HAPA Theorem (Hyper Alkalic-Proofmood Theorem)

Alkalic-Proofmood nonHyperVersion형식증명의 근거.

동시에 유일한 Alkalic-Proofmood HyperVersion에서의 형식증명

`y = x`, `y = z`가 가설일때,

규칙 `using x = y → (Φ ↔ (Φ [x := y]))`를 통하여, `y = z ↔ x = z`를 보인다, 즉,

문법상, `y = z ↔ (y = z ↔ (y = z [y := x]))`= `y = z ↔ (y = z ↔ x = z)`이므로,

y = z ↔ (y = z ↔ x = z)를 표현하기 위한 잉여적인 체계다.

Alkalic-Proofmood (Power Up - Version)

증명에 앞부분에 붙여야 할 한정사가 추가되었다, 부분증명을 만들어서 중첩 가능하기에, 각 기능을 동시에 붙일수 없다.

- AristotelProof(비 명시시 기본) : 기존 증명 방식으로 증명

 - DavidHumeProof : Φ□에 대해, t번 라인마다 매거적 귀납법을 쓰고, 옆 열에는 Φ□가 귀납법 증명에 쓰이는 경우, 쓰는 칸이 된다. 맨 마지막줄에, 번호 없이, 귀납법 증명의 종류를 기재할때, `∴ Φ□, Φ□₊₁, ..., Φ□ ⊨ Φ□₊₁` (강함), `∴ Φ□ ⊨ Φ□₊₁` (약함), `∴ Mod(Φ) = ℕ` (일반적인 수학적 귀납법) 으로 기제한다.

 - EuclidianProof : 귀류법 (HegelianProof랑 다르다, 귀류법이다) ; 반증 마지막에, `∴ ⊥ ∴ ⊨ ¬

Φ ∴ Φ`를 놓는다. (`¬Φ`는 결론을 뜻한다.)

 - HegelianProof : 반증 (결론이 부정이 나오므로, "결론이 아니다"를 증명할때 쓰임; 왜냐하면, 기존 버전에서는 논리적 오류가 나오면 오류위치를 지적하고 프로그램이 종료됐기 때문에, 오류를 만들어 반증한 후, 종료하지 않는 AristotelProof가 필요했음)

또한 검증 프로그램 정지를 피하기 위해,

- PreviewVersion : 이 부분•전체 증명에 대해서, 프로그램 정지 후 오류 지적을 제외하고, Preview리스트에 추가한다, 근거 없는 부분이라, 이걸 단 증명을 참고해서 에러나면, "Referance on Preview"에러 로그를 따로 뱉은 후 평상시 에러처럼 에러난다

- DebugVersion : 오류가 나는대로, 디버그를 해주며, 훑고 지나간다, **프로그램 전체에 적용된다.**

- ConjureVersion : 추측으로써, 정지를 피할곳에, `�`를 삽입한다, 이 부분•전체 증명은 가설(Hyperthesis)로 취급된다.

- NormalVersion (비 명시시 기본) : 기존 방식

마지막으로, 다항식의 계산을 원활하게 하기 위해,

`Polynomial Simplify`라는 부분증명 폼을 넣고 다음을 인수분해하거나, `P(x) = 0`꼴을 풀면 (후자는 미리 `using P(x) = 0 Algorithm`이라고 명시) 오류 없이 증명을 받아들여준다.

A. `LinearSimplify`명령을 통해, LinearSimplify Theorem에 근거하여, 미지수가 여러개인 일차식을 정리한다

B. `Substracting [y := $x^n$]`명령을 통해, x^n을 y로 치환한 문장 `Φ`에 대해, `Substracting Variable`필드에 넣은 참인 문장 `y = $x^n$`에 따라서, Φ [y := x^n]가 나올때까지, 미리 일차식마냥 치환한 상태로 작업하게 해준다. (치환 변수 필드 논법; `Substracting Variable Field Proofs`)

C. `Solution (a, b, c, d, e)`명령을 통해, 2차 ~ 4차식을 인수분해(근의공식) / 전개(비에트의 정리)한다.

D. `TschirnhausTheoremSubsituate (n, a, b, x)`명령을 통해, `[x := t + b/na]`를 적용한다, 마찬가지로 증명의 원활함을 위해 Substracting명령에 근거한다 (사실 그럴 필요도 없이 구문론적으로 연산자를 정의해도 되는 간단한 문장(`[x := t + b/na]`)이지만)

E. 부분증명 문법에서 `synthetic division` 한정사로, 조립제법 이용 (생략표기가, 매거적 귀납에서 고정된 열의 다수의 행에대해 쓰이므로, 여기서는 쓸때, 행을 다항식으로, 계산 과정순이 열로 되므로, 플려화야하는 단점이 있다.)

F. `Alright synthetic division`한정사로, 일반적인 조립제법을 쓰고, 전처리 단계에서 synthetic division로 컴파일

G. 분배법칙을 위해서, `distribute[ 대상 ]` 안에 전부 넣어가지고, 이 증명 시스템용으로 있는 `분배법칙의 일반화 정리`에 따라, 분배함

H. `AlgebraicFormula` : 미리 증명한 곱셈공식을 이용해서, 계산되었음을 명시한다.

I. Gaussian Eimination or ERO & Subsituate : 가우스 소거 혹은 가감/대입

J. System of Quadratic Equations by Quadratic Form : 이차형식으로 연립이차방 풀이

K. System of Quadratic Equations by Cubic Form : 삼차형식으로 연립삼차방 풀이

L. Règle de Cramer : 크래머의 공식으로 풀이

M. `Extraneous Root is (□)` : 무연근 명시

N. `PolynomialFractionize` : 다항함수 분수화

O. `SolvePolynomialFraction` : 해당값 풀이

P. `Fractions Solution is (□)` : 해 명시

형식증명의 오토마타용 문법

`□.` 라인이 부분증명이라면, `□.line.`식으로 라인을 표기한다.

그리도, 라인과 라인 사이에 오직 whitespace및 `-`,`-`,`—`만 있을경우 해당 라인을 가독성 용도로 보고 주석처리한다.

또한, line표기에 앞선 점찍은 부분 앞에서 `|`부분이 문자열의 특정 열마다 이어지고, 끝나는 말단이 앞서 설명한 `-`꼴의 주석에 연결되어있다면, 해당 부분도 따로 오류처리하지 않는다.

그외에는, 열 구분자이기에 주석으로 보지 않는다

마지막으로, `[NOTE : ]`형식을 주석으로 본다.

마크다운 문서 내부에 위치했다면, HAlkalic-Proofmood(Hyper Version), Alkalic-Proofmood(일반 버전), PowerAP(Power Up버전)으로 코드 이름인 부분만 읽는다. 또한, 마크다운 부분에 부분증명 코드부분은, 부분증명으로 렌더링한다.

마지막으로 그렇게 html화되어 정리된 렌더링 뷰는, LaTeX 표기 기능을 추가해야만 할것이다.

(그럼에도 해당 html뷰는 아직 형식증명 검토가 안돌아갔으므로, 컴파일 상태인거지, 실행 상태가 아니다. 실행은 실행기에 돌려야, 문서 내부를 파싱해서, 부가적으로 제공된, [labare](https://faraway6834.github.io/unbeauty/privateNote/Proof/labare)•[unbare](https://faraway6834.github.io/unbeauty/privateNote/Proof/unbare) 코드와 함께 해석하여(labare•unbare는 인터프리터 언어가 아니고, 정형 대이터 겸 사용자 편의 대이터 겸 Low Level 컴파일 언어다.), 검토된다; 이제보니 실행기보다는, 형식증명 검토기라는 명칭이 더 적합하다, 프로그래밍 언어는 하나도 실행하지 않고, 추론규칙을 재대로 활용했는지만 검사하여 검토작업(오류나 로그나 상태 표시)만 하기 때문이다.)

논리적으로 다룬다 전재할때, 대수식은 논리적으로 그 뜻이 해석 • 계산된다.

그렇지 아니하면, 논리적 해석 흐름에서 논리기호가 도출될수가 없다.

식의 계산은 그 값의 배정인 $(\bar{x}, f(\bar{x}))$와 같이 이루어지는데, 이 방식을 거부하는것은, 논리를 쓰지 않겠다는 말과 같다. (장자 왈 갓나서 죽은 아기보다 오래 산 사람은 없으니 팽조(760살이 넘게 살았다는 전설 상의 신선)도 일찍 요절한 사람)

대수식의 논리적 해석 흐름중 논리적 귀결관계의 도출

Step.1. 방정식을 만족하는 집합으로써의 모델집합이 해집합임을 보이자

먼저, 다음과 같은 다항식 함수 P를 정의하자.

> `P :≜ λA. λx. Π□ x - A□`

그리고 다음과 같은 방정식화 논리함수 Φ를 정의하자.

> `Φ :≜ λf. (f(x) = 0)`

그리고 마지막으로, 다항 방정식 ㅍ을 정의하겠다.

> `ㅍ :≜ φ • P`

그러면,

> `Mod(ㅍ(A)) = {x | x ⊨ (Π□ x - A□ = 0)} = {x̄ | Π□ x̄ - A□ = 0} = {A□ | ∀i}`

임이 당연하다.

Step.2. 논리적 귀결관계의 도출

다항방정식 표(A), 표(B)에 대해,

0. Mod(표(A)) ⊆ Mod(표(B))

1. $\{A_i \mid \forall i\} \subseteq \{B_i \mid \forall i\}$

2. $\forall i\ A_i = B_i$《주의 : 비약이다, 저건 배열을 정렬해야만 성립한다.》

3. $\exists C\ P(B) = P(A)P(C)$

4. $P(A) \mid P(B)$

으로,

다항식 f, g에 대해 다항방정식 $\Phi(f) \models \Phi(g)$

이면이

$f \mid g$

대수식의 논리적 해석 흐름중 진리값 배정되는 명제논리 결합자의 도출

¬x = T - x로 해석됨을 보이자. (경고 : 형식증명 아님)

A. proof of `x ≠ T ⊢ T ± x ≠ (1 ± 1)T`

0. `x ≠ T` (비 귀류법식 전제 문장)

1. `T ± x ≠ T ± T` (이항 by 함자 `(T ±)`)

2. `T ± x ≠ T ± T = (1 ± 1)T` (1번의 연장선에서 계산)

3. `T ± x ≠ (1 ± 1)T` (2번에서 식 요약) … ■

이어서

B. proof of `⊨ (1 + 1)T = 0 ∨ (1 + 1)T = T`

0. 먼저 part A by `⊨ (1 + 1)T = 0`와 part B bt `⊨ (1 + 1)T = T`로 나눠서 생각하자.

1.A. (1 + 1)T = 0 (귀류법식 전제 문장)

2.A. (1 + 1)T = 2T = 0 (1.A번의 연장선에서 계산)

3.A. T = 0 ⊨ 2.A. ⊨ (1 + 1)T = 0 (연역)

4.A. ⊭ T = 0 ⊨ 2.A. ⊨ (1 + 1)T = 0 (연역)

5.A. ⊭ (1 + 1)T = 0 (연역) … ⊥

6.A. ∴ ⊭ (1 + 1)T = 0 (연역) … ■

1.B. (1 + 1)T = T (귀류법식 전제 문장)

2.B. (1 + 1)T = 2T = T (1.B번의 연장선에서 계산)

3.B. T = 0 ⊨ 2.A. ⊨ (1 + 1)T = T (연역)

4.B. ⊭ T = 0 ⊨ 2.A. ⊨ (1 + 1)T = T (연역)

5.B. ⊭ (1 + 1)T = T (연역) … ⊥

6.B. ∴ ⊭ (1 + 1)T = T (연역) … ■

이이이어서

C.1. A, B ⊨ (x ≠ T ⊢ T - x ≠ (1 - 1)T) (A, B 번에서 귀결)

C.2. A, B ⊨ (x ≠ T ⊢ T - x ≠ (1 - 1)T = 0T = 0) (C.1. 번의 연장선에서 계산)

C.3. A, B ⊨ (x ≠ T ⊢ T - x ≠ 0) (C.2. 번에서 식 요약)

C.4. A, B ⊨ (T - x = 0 ⊨ x = T ⊨ x) (C.3. 번에서 연역추론 : 대우) 《주의 : "x = T"의 의미를 "진리값 x가 참"관점으로 해석함.》

C.5. A, B ⊨ (T - x = 0 ⊨ x) (C.4. 번에서 식 요약) 《주의 : 근거인 C.4.에서 "x = T"의 의미를 "진리값 x가 참"관점으로 해석함.》

C.6. C.5. 번 내용 ⊢ ¬x = T - x (최종결론) 《주의 : "x = T"의 의미를 "진리값 x가 참" 관점으로 해석함.》

Q.E.D.

이 ㅣ ㅣ이이이이이이이이이이이 ㅣ 이이잉ㅇㅇㅇ이이이서 ㅓ

x ∧ y는 xy로 해석됨을 보이자.

T에대한 방정식 (T - x)(T - y) = 0의 해는

x = T ∨ y = T이다.

따라서, x = T ∨ y = T ⊨ T - (T - x)(T - y) = T고,

x ∨ y = T - (T - x)(T - y)로 해석된다.

이때 De Morgan's Law, ¬(¬x ∨ ¬y) = x ∧ y서

T - T + (T - T + x)(T - T + y)

= xy이다.

… Done.

방정식의 의미 : 술어논리(함수논리)의 술어로써, 잠정적으로 특칭양화사를 사용해, 잠재적으로 전칭양화사를 사용함

방정식 $P(x) = 0$이 불능이란것은

$\nexists P(x) = 0$란 뜻이며

$\forall P(x) \neq 0$이란 뜻이고 … ①

방정식 $P(x) = 0$가 불능이 아니라면

$\exists P(x) = 0$이다. … ②

방정식 $P(x) = 0$이 부정이란것은,

부정방정식이므로,

$\forall P(x) = 0$이다. … ③

①에서, 불능형 방정식 $P(x) = 0$에 대해,

$P(x) \neq 0$은 부정형이고, … ④

부정형 방정식 $P(x) = 0$에 대해,

$P(x) \neq 0$은 불능형이다 … ⑤

그렇다면 ③에 따라 다음을 정의하자,

$`\Phi :\triangleq \lambda f.\ (\exists f(x) = 0)`$

$`P :\triangleq \lambda f.\ (\exists f(x) \neq 0)$

그러면 다음을 알수 있다.

이이이이이이이이이이이이이이이어서

④에 따라, Φ(f)가 거짓 이면이 P(f)는 부정형

⑤에 따라, Φ(f)가 부정형 이면이 P(f)는 거짓

Φ(f)가 참 이면이, f(x) = 0를 만족시키는 x존재

P(f)가 참 이면이, f(x) = 0을 불만족시키는 x존재

부정형 방정식을 만들고 싶다? 하면

¬Φ(f) = P(f), Φ(x) = ¬P(f)에서,

불능형 방정식 Φ(f)에 대해 부정하거나,

불능형 방정식 P(f)에 대해 부정하면된다.

술어 P에 대해

Mod(P) = ∅ 이면이 ∄P(x) 이면이 ⊭ P

이면이

Mod(¬P) = U 이면이 ∀¬P(x) 이면이 ⊨ ¬P

따라서, 방정식은 기본적으로 특칭 술어로써, 사용할수 있음

PART 3. 방정식이 쉽다는 근거

SIOT[시옷], Standard IO Table (Mean : Standard Input and Output Truthtable)

교육 구조.

논리 연산을 [FAN, Function AbbreviatedNaming](https://faraway6834.github.io/unbeauty/privateNote/functional_petterns_commandstyle_naming#%EA%B8%B0%EB%8A%A5%20%EC%B6%95%EC%95%BD%EB%AA%85%EB%AA%85%EB%AA%85\(%EC%B6%95%EC%95%BD%EB%AA%85%20%EB%AA%85%EB%AA%85\)%20\(Function%20AbbreviatedNaming\)%20%EC%B2%B4%EA%B3%84%20%EC%84%A4%EB%AA%85)으로 정의하여,

기능으로 사용하는 도구로 봄.

SIOT의 해석.

그리고, SIOT가 그 모든 연산을 나타내는 계산표 (계산표의 예시론 덧셈표와 곱셈표와 제곱근표와 상용로그표가 있음)임을 보임.

논리 연산의 해석.

기존에 배운 논리 연산과, 그 예시로 든 글들을,

활용할때, 단순히 FAN으로 작성한 축약명 연산자들을 SIOT를 들어서 동등함을 보인다.

그러면 활용시 동등하니까 그냥 쓰면 된다.

논리 연산이라는 명칭의 납득.

이분법적 논리학이 극단적이라고 디스하고싶지만,

원래 수학이 그렇다고 하자.

슬슬

SIOT의 해석.

그리고, SIOT가 그 모든 연산을 나타내는 계산표 (계산표의 예시론 덧셈표와 곱셈표와 제곱근표와 상용로그표가 있음)임을 보임.

논리 연산의 해석.

기존에 배운 논리 연산과, 그 예시로 든 글들을,

활용할때, 단순히 FAN으로 작성한 축약명 연산자들을 SIOT를 들어서 동등함을 보인다.

그러면 활용시 동등하니까 그냥 쓰면 된다.

논리 연산이라는 명칭의 납득.

이분법적 논리학이 극단적이라고 디스하고싶지만,

원래 수학이 그렇다고 하자.

모양세

사실은 진리값배정용 칸부분을 Input이라 명명하고, 표 왼쪽이 위치시키고, 나머지 연산을 오른쪽에 위치시킴.

그리고 진리표 윗부분에 Input과 Output이라는 대분류 셀만 덧붙침.

연산자의 좌측과 우측 (피연산자, 진리값 배정 대상)

좌측과 우측이 연산 대상이고, Left Side와 Right Side라고 영어로 말한다.

각 명칭을 줄여서 LS와 RS라고 부른다 (두문자어)

얍

참과 거짓

참과 거짓은 특수한 정보로,

컴퓨터에서 이 의미를 나타내기 위해서는 정보를 저장해야한다 (팁 : "사실 프로그램으로 때워도 되" 라고 농답식으로 납득시킬수 있음)

컴퓨터에서는 참 = 있다 = 1, 거짓 = 없다 = 0

연산자명 명명 규칙

L□R■는 L이 □고 R이 ■라는 뜻이다.

명제 (판별가능) 이고, 진리값이 들어감으로, 의미로써 연산값을 가지고, 표기 가능하다.

0000이 정규형식일때 : AZVC(AlwaysZeroValueCommand)

1100이 정규형식일때 : GRSV(GetRSValue)

1010이 정규형식일때 : GLSV(GetLSValue)

정규형식의 보수를 부정으로 이해할때 : Non접두사 사용, 약어 `N`

이외의 연산자들은 논리적으로 유도한다.

그부분에선 이해 못해도 된다. 결과만 사실 쓰면 되고, 나중에 배워도 활용엔 문제없으니.

CMD 대수 : 1. CMD 차원

M 연산

- M 장의 각 함수로, 언커링된 D 연산임.

M 구성

- D 공간에서 변환되어 구성.
- M 장에서 텐서곱 바인딩하여 D공간을 결정함.

M 표기

- M 표기 정의식에, 각 M 기저에 대하여 대괄호를 씌어, 텐서곱 바인딩과, M표기 정의식에 대한 탠서곱 바인딩을 이룸.

M 대응

- D 장에 항상 대응하여 M장을 항상 D장에 대하여 존재하게 하는것.

C 공간과 그 체계

- 슬롯임.
- M과 D를 가짐 (눈치 챗겠지만 CMD, 커멘드)
- M대응과 D대응을 적절히 수행하는 C대응이 존재.
- C정의를 할때, M과 D의 수정이 이루어지고, C대응으로 C내부공간(사실 이게 지금 말하는 부분)의 수정이 이루어지는 C 구성이 존재.

C 표기

- 각 정의의 추가시 C 구성을 이루게 하는 상위차원에서 평가하여 C 슬롯안에서 호출할 내부 아웃소싱 기계. (구성)

CMD

2. CMD 심볼 표기

- □수열의 인덱싱을, □의 상단의 선의 우측지점의 선분을 제외한 부분을 좌표평면위의 그래프로 부분적으로 정의하고, 나머지 부분에 글자를 넣도록 정의하여, 기호용으로 그래프로 그려진 형태의 심볼을 표기법용 기호로 쓰게하는것.

- CMD 이용 : 괄호없이 그저 폴란드 표기법으로 쓰는것

(표기)

3. CMD 포함.

최소한의 방법으로

- A. 정의하는 부분의 차원 낮추기

- B.1. 최소한의 의미적으로 최적이동한 포함.

등등으로 CMD평가용으로 상위에 구성할 차원을 의미적 구성용으로 치우는것. (알고리즘)

기능 축약명명명(축약명 명명) (Function AbbreviatedNaming) 체계

CMD대수로 함수형 연산자를 선언하는것을,

기능이란 뜻의 Function에서, Function역할을 하는 빈칸에 대한 패턴의 식으로, 연산자를 축양명으로 명명하는 개념으로 배울수 있는 사용체계.

기능 축약명명명 표기

바인딩 표기가 아닌 `#`이후 등식으로 작성한다.

기능 축약명명명(축약명 명명) (Function AbbreviatedNaming) 체계 설명

번호가 매겨진 빈칸이 있는 기능 패턴에, 기능 축약명 연산자의 특정 번째 입력값을, 번호가 메겨진 빈칸안에 대응시켜, 단순히 축약식의 축약을 풀어 그 뜻으로 치환하여 해석하도록, 기능 축약명 연산자를 기능에 대한 축약명으로 정의하도록 하는 기능 축약명을 명명하는, 기능 축약명 명명 체계를 만드는것.

따라서 의미적으로 등호가 성립한다.

전산표기

빈칸에 번호를 상단에 메길수 없기에,

`🅰🅱🅲🅳🅴🅵🅶🅷🅸🅹🅺🅻🅽🅼🅾🅿🆀🆁🆂🆃🆄🆅🆆🆇🆈🆉 ` 와 같은 글자를 이어서 번호 빈칸을 대체한다.

예시 : Trinity 기능 축약명 연산자

(참고사항 NOTE : markdown에 소스타입을 FAN으로 명시함, 겹치는게 있을지도 모름)

정의

```FAN

# Trinity = 🅰(🅱 - 🅲) + 🅲

```

사용 (값은 여기서 12가 됨)

```FAN

Trinity 1 12 21

```

LAFTF 1.1Logic-Algebra Formular TransForm 1.1 (구버전 불완전)

def)

⌈ To : {x|Set(x)} × {x|Set(x)} → {z|z={f:x → y | p(x, y)}}

p: |

{

⌊ To : x, y ↦ {f | f : x → y}

def 표기) A To B ≡ To(A, B)

def 표기) A To B ≡ A 2 B

---

def) (∃! int ∈ Bool2{0, 1} ∃! int^{-1})(int^{-1}(x) ≡ (x = 1) ≡ (x ≠ 0))

def) bool ≡ int^{-1}

기호 유도

(bool · f)(x, y) :≡ bool(x) ∧ bool(y)

(x̄, f(x̄)) = (x̄, ×(x̄)) (단. <×, Y_i□□>)

∵ ⎧ f(0, x) = 0 = 0 × x

⎪ f(1, x) = x = 1 × x

⎨

⎩ \[x, y\]□ = 1 = \[x, y\]$_x$ (h=f, 기호의 한계…)

∵ f(n, x) = f(x, n) (단. n ∈ Y_i□□)

∴ f(x, y) = xy

단순히 같다는 의미이고…,

수학에 생각이란게 유효하다면 이건 자명한게 아닐까 조심히 추측해본다.

-

(bool · f)(x) :≡ ¬ bool(x)

f(0) = 1, f(1) = 0

∴ f(x) = (0-1)/(1-0) x + 1 = 1 - x 로, 그래프 위에서 점을 찍을때, 우리가 보고자 하는 점의 위치를 f(0)에서 f(1)로 수를 바꿔보면, 감소하는 수는 1을 얻으니 당연하다.

(아니 생략이 ㄹㅇ ㄹㅈㄷ, 어짜피 요약본이긴 한데 ■겁네)

그러면 Conjuntive Normal Form에 따라

⎧ b = ∑ A□2^{n-1} (n=1~4, 단. $\bar{x}$ = ((0, 0) ∧ $f(\bar{x}) = (A_1, A_2, A_3, A_4)$)

⎪ (0, 1)

⎪ (1, 0),

⎨ (1, 1))

⎩ Hexf(b) = f 인 Hexf(b)는 xy와 1-x로 조합 가능하다

이때, $x \oplus y = (x \vee y) \wedge (\neg (x \wedge y)) = (x \wedge \neg y) \vee (y \wedge \neg x)$ 인데,

$int(x \oplus y) = int(x \vee y) - int(x \wedge y)$

∵ $f_1(x, y) = xy$

$f_2(x, y) = x + y - xy$

$f_3(x, y) = int(bool(x) \oplus bool(y))$

$Hexf^{-1}(f_3) = Hexf^{-1}(f_2) - Hexf^{-1}(f_1)$

또한,

$int(x \oplus y) = int(x \wedge \neg y) + int(\neg x \wedge y) - int(\neg x \wedge y)\ int(x \wedge \neg y)$

$int(\neg x \wedge y)\ int(x \wedge \neg y) = int((\neg x \wedge y) \wedge (x \wedge \neg y)) = int(\perp) = 0$

시프트

대충 프랑스에서 아이디어 떠올려서 영적인 힘을 줘서 고맙다는 ■소리가 적힐 칸이었는데…

■■이런또■이같은인생아지■난■가타서■나못참아죽■네■친어떻게■■내가■고생해서얻은LAFTF1.0이■■이미있냐위키백과에서xor기호얻으려고■■들어가봤더니나중에찾는건다른곳에서했고■벌아진짜■같이영문위키백과에■■아니거기를먼저들어갔더니얻은기호가아니진짜■얼탱이가없어가지고하아 제칼킨 식이라고 ■ㄴ좋은 이론이 선행으로 있었는데 이거 모르고 위에 내용 ■■ 혼자 하나하나 유도한 내가 ■■이지 ■■이야 ■나 ■생 ■반 ■해봤네 ㅋㅋㅋ ■친 뭐 앞으로 서술할 비트연산도 마찬가지겠지 뭐 하 ■발 인생.

shl□(x) = shl□(2□¹ + x) (β = B (Bits) 동일하게, 기호의 한계, 참고로 겹쳐가시고 ■벌 한번 더 a서 β로 수정함. 아이고야 ㅠㅠ)

(shl□)□□ (Λ ≡ lm, 동일한 한계로 이런식으로 표기하겠음, ‘*’)

(shl□)□ (아랫첨자 a가 없는 관계로… ■벌 ‘*’)

원리; 욕 작작해 나자신 검열을 해도 상스럽네 수준떨어져

원래는 (=필기 원문대로 쓰면은) “그런대 아름다운 사실,” 이라고 써야하는데 ■같네 기분이 아직도 ㅋㅋ 이런 ■■ 아■다운 새상

$f(x) = x - \lfloor x \rfloor$

$f(x) = f(1 - x)$

f□□ = 1 (참고 : ⁽*⁾)

$f_a = 1$ (참고 : ⁽*⁾)

$g(x) = 2f(x)$

g□□ = f□□, $g_a = 2f_a$, f□□ = f_a,

g□□ = k, $g_a = 2k$ ($k = f_x$)

이어서

(shl_β)□□ = k, $(shl_\beta)_a$ = 2k ($k=2^{B-1}$)

$shl_\beta(x)$ = k g(x/k)

또한, 동일하게 그래프나 함수 버전, 방금전 버전 유도도 있지만, 단순하게도, shl = ⌊x/2⌋ 다. x ÷ 2 = y … r, x ≡ y (mod. 2)고, ⌊x/2⌋ = (x-r)/2 다. (아 아래로 …화되는거 ■ㄴ 띄껍내개■■ (딥■))

시프팅시 일부 비트의 정보 손실(?) 을 이용하여

$idx_\beta(x) = shr^B \cdot shl_\beta^n(x)$

—

요약 나중에 하겠음 (EZ)

—

def) bitwise f_β ($\Sigma A_{1,i}\ 2^{i-1}$, …(×n)..., $\Sigma A_{□,i}\ 2^{i-1}$) ≡ $\Sigma\ f(A_{1,i}$, …(×n)..., $A_{□,i})\ 2^{i-1}$ (i = 1 ~ B)

—

nagation = bitwise-no, $nagation_\beta(\Sigma A_i\ 2^{i-1} = (= x)) = \Sigma\ (1-A_i)\ 2^{i-1} = \Sigma\ 2^{i-1} - \Sigma\ A_i\ 2^{i-1} = (2^B\text{-}1) - (\Sigma A_i\ 2^{i-1} = (= x))$, $nagation_\beta(x) = (2^B\text{-}1) - x$ (i = 1 ~ B)

—

p(◦, *) := (a ◦ (b * c) = a ◦ b * a ◦ c인 (◦, *))일때

p(×, +)은 맞는데, p(+, ×)은 아니라서, 다항식 f에 대해 $f(\Sigma A_{1,i}\ 2^{i-1}$, …(×n)..., $\Sigma A_{□,i}\ 2^{i-1})\ 2^{i-1}$를 보통 구할때, 부정 연산이 아닌 논리연산은 항을 교환 불가능하다.

ㅋㅋ

킹치만 비트 인덱싱좌가 있다고 코삣삐.

bitwise $f_\beta(x, y) = \Sigma f(idx_\beta(B - i, x), idx_\beta(B - i, y))2^{i-1}$ (i = 1~B)

산술연산으로 다 1초컷 ㅋㅋ

참고 : f(x) = x - ⌊x⌋ 는 톱니파의 일종으로, actan(tan(x/π))도 톱니파의 일종으로, 가우스 함수는 ㄹㅇ로 그냥 ■ㄴ 초딩도 이해가능한 ■ㄴ 자명한 산술연산이라고 ㅋㅋㅋ

—

def) (∃! boolf ∈ {x | Set(x)}2{f:U→{0,1}} ∃! $boolf^{-1}$)($boolf^{-1}$(p) ≡ {x | p(x)})

def) setize ≡ $boolf^{-1}$

헛소리; 개쪽; 의의는 기계나 계산기 진리표 공식 이렇게 5개

Laftf 1.1 (이거)를 쓰는 튜링 머신에서 작동하는 명제 p는

산술, 논리, 비트연산에 대한 명제일 수 있고, setize(p)는 집합이고, 결국에 집합은 S = {a_1, …, $a_□$} 일시 x = a_1 ∨ … ∨ x = $a_□$ 일 수도 있어 모든 집합 S는 boolf(S)로 p로 환원가능함으로, 집합은 여기서 말한 전체 집합의 모임의 집합에 한정해서 다룰수 있다.

사실상 명제 =(= 방정식)임으로 말 다했음, 그냥 ■ㄱㄴ

여기까지가 LAFTF 1.1이었다 ㄱㅅ

instagram @leenuxmathno7e

[github.com/FarAway6834](https://www.google.com/url?q=http://github.com/FarAway6834&sa=D&source=editors&ust=1737545183319986&usg=AOvVaw0udQDDfIxTu4hUrjAnk67S)

/c0dk1ddy

\[번외\] ANF (산술 정규 형식), Zhegalkin 수식, 그리고 그를 응용한 비트연산을 모듈러-2 환이 아닌, 진리값 배정에 따른 연산자를 함수로 해서 그 함수만 연산으로 사용한 모듈러-2 환으로써, 내부적으로는 모듈러-2로 정리되지 않치만 추상화해서 컴퓨터에서 사용하기에는 LAFTF가 더할나위없이 좋은것 같다.

유튜브에서 무슨 티타늄합금이 스스로 수학문제를 생각해서 푼다는 헛소린지 군침이 싹도는 진짜 소식일지 모르지만 거기에 쓰였으면 좋겠다.

PART 4. PROGRAMMING FW

unbeauty; 유사한걸 이미 정보 수행으로 (수열을 이용하고, 귀납적 정의 : 제귀등; 쓰므로 **이게 수열파트(발표 핵심 명분)**)

unbeauty esolang (0^0=1, x->0 x^x->1)

mathmatics.... beautiful 4 me....

`sementically-recursive-defined-recurrence-formular based` programming language

actually, I didn't finished this work, because of my high school exam.

~~[redacted]ing Korea~~

example : (unbeauty에 의존 + 선택구조 공식)

noptlib.unbe

```unbeauty

sqrt = cacher[1](λx. x^0.5)

abs = cacher[1](λx. this.sqrt(x^2))

partial_beq0c = cacher[1](λx, n, p. ((0^this.abs(n)) + n×(x + ((-1)^p)×n) ÷ (0^this.abs(n) + n^2)) × f(x, n - 1, p))

partial_beq0 = cacer(λb, x, p. this.partial_beq0(x, 2^b - p, p))

beq0 = cacher[1](λb, x. this.partial_beq0(b, x, 0) × this.partial_beq0(b, x, 1)

beq = cacher[1](λb, x, y. this.beq0(b, this.abs(x - y)))

dose_it_positive = cacher[1](λb, x. this.beq0(b, this.abs(b, x - this.abs(x))))

__cmp__ = cacher[1](λb, x. this.beq(b, x) + (-1)^(this.dose_it_positive(b, x)+1))

__le__ = cacher[1](λb, x, y. this.dose_it_positive(b, x - y))

conditionalidx = cacher[1](λp,x,y.p×(x-y)+y)

conditional = cacher[1](λp, x, y. p×x + (1-p)×y)

_4bit_eqer_ = cacher[1](λx,y.this.beq(4,x,y))

_4bit_idxer = cacher[1](λi,a0,a1,a2,a3,a4,a5,a6,a8,a9,aA,aB,aC,aD,aE,aF.this._4bit_eqer_(i,0)×a0+this._4bit_eqer_(i,1)×a1+this._4bit_eqer_(i,2)×a2+this._4bit_eqer_(i,3)×a3+this._4bit_eqer_(i,4)×a4+this._4bit_eqer_(i,5)×a5+this._4bit_eqer_(i,6)×a6+this._4bit_eqer_(i,7)×a7+this._4bit_eqer_(i,8)×a8+this._4bit_eqer_(i,9)×a9+this._4bit_eqer_(i,10)×aA+this._4bit_eqer_(i,11)×aB+this._4bit_eqer_(i,12)×aC+this._4bit_eqer_(i,13)×aD+this._4bit_eqer_(i,14)×aE+this._4bit_eqer_(i,15)×aF)

짤림
```

짤려서 이어짐

not = cacher[1](λx.1-x)

and = cacher[1](λx,y.x×y)

sor = cacher[1](λs,x,y.x+y-(1+s)*this.and(x, y))

or = cacher[1](λx,y.this.sor(0,x,y))

xor = cacher[1](λx,y.this.sor(1,x,y))

nand = cacher[1](λx,y.this.not(this.and(x,y)))

nor = cacher[1](λx,y.this.not(this.or(x,y)))

nxor = cacher[1](λx,y.this.not(this.xor(x,y)))

sub = cacher[1](λx,y.this.and(x,this.not(y)))

rsub = cacher[1](λx,y.this.sub(y,x))

rimp = cacher[1](λx,y.this.not(this.rsub(x,y)))

imp = cacher[1](λx,y.this.not(this.sub(x,y)))

a = cacher[1](λx,y.x)

b = cacher[1](λx,y.y)

nota = cacher[1](λx,y.this.not(x))

notb = cacher[1](λx,y.this.not(y))

logicalerr = cacher[1](λx,y.0)

alwaystruth = cacher[1](λx,y.1) 아니 또

하,,,

lpu = cacher[1](λcod,x,y.this._4bit_idxer(cod,this.logicalerr(x,y), this.and(x,y), this.sub(x,y), this.b(x,y), this.rsub(x,y), this.a(x,y), this.xor(x,y), this.or(x,y), this.nor(x,y), this.nxor(x,y), this.nota(x,y), this.rimp(x,y), this.notb(x,y), this.imp(x,y), this.nand(x,y), this.alwaystruth(x,y)))

__gt__ = cacher[1](λb, x, y. this.not(this.__le__(b, x, y)))

__lt__ = cacher[1](λb, x, y. this.__gt__(b, y, x))

__ge__ = cacher[1](λb, x, y. this.__le__(b, y, x))

__ne__ = cacher[1](λb, x, y. this.not(this.beq(b, x, y)))

bits2bool = cacher[1](λb, x. this.not(this.beq0(b,,x)))

shr = cacher[1](λx, n.x÷(2^n))

shl = cacher[1](λb, x, n.(2^b)×x-(2^b)×(x÷(2^(b-n))))

idx = cacher[1](λb, x,i.this.shr(this.shl(b, x, i), b))

bpucc = cacher[1](λb, cod,i,x,y.this.lpu(cod, this.idx)(b, x, i), this.idx)(b, y, i)) × (2^i))

bpuc = cacher[1](λb, cod,i,x,y.this._bpucc_(b, i,x,y) + this.bits2bool(i)×this._bpuc_(b,i-1,x,y))

bpu = cacher[1](λb,cod,x,y.this._bpuc_(b,cod,b-1,x,y))

```

ex2.ubt - extend not base cls, ex1 cls.

```

:noptlib

fibo = cacher[1](λb, x.this.conditional(this.beq0(b, x), 0, this.conditionalidx(this.beq(b, x, 1), 1, this.fibo(b, x - 1) + this.fibo(b, x - 2))))

```
```

어음

[coding plan 24.02.24 ~ ...](./plan)

compile time lazy evil optimizing

`[○×]this.__NAME__(□)` -> `([○×]this.__NAME__)(□)`

 > also as `conditional` too. (when it is arguemnt, not return)

FUOIR Unbeauty Optimize IR

example.unbe

```unbeauty
temp = cacher[0](λx. 2 × x)
main = cacher[0](λx. this.temp(x) + 1)
```

어어므

example.auty (jsonic)

```fuior
[
	0,
	"x",
	"this.temp(x) + 1",
	{
		"temp" : [0, "x", "2 × x"],
	}
]
```

plan change

not using macro, will use PCRE

symopt

just add optimizer at ir,

that sympy optimizer.

`this.□(○)` -> `base64(hash(□(○)))` to symbolize (like-lexing)

LinearUnbeauty 계획 (짜피 내 코딩실력은 뭐든지 만들줄 아는 최적화쟁이니 걱정 ㄴㄴ)

LinearUnbeauty 계획

선형대수를 지원하는 [Unbeauty](https://faraway6834.github.io/unbeauty

... 가 끝임

labare 언어 계획

[Linear Unbeauty](https://faraway6834.github.io/unbeauty/privateNote/Proof/LinearUnbeauty)의 확장이며, 람다 산술이 지원된다.

구체적 구현은 미룰 예정이며,

성능대신 생산성이 높아보이는데, 실제로는 내 성격상 성능쪽으로 갈것같다.

형식증명에 쓰일 언어이다.

행렬 입력 / 반환을 언커링된 Input / Output으로 처리하는, Subrootine Realationship이라는 종류의 연산이 있다만, 실제로는 흉내만 내고, 아무짓도 안한다.

Proofmood 모드에서만 쓰는 기능이다

명칭 [유래](https://genius.com/Edna-st-vincent-millay-euclid-alone-has-looked-on-beauty-bare-annotated)

음...

unbare 언어 계획

형식증명용 언어이다.

실행 목적이 아닌 [labare](https://faraway6834.github.io/unbeauty/privateNote/Proof/labare)코드를 뱉는 용도이다.

output laber code의 구조 :

- virtual pararel scadular
- calculatible part
    - pararel scadular
    - haltible part
    - outof haltible part
- uncalculatible part

그러니까, 수학적인 Subrootine-Relationship코드를 만드는게 목적이다.

PART 5. 그래서 뭐

「Alkali Mathmatics with easy explanation by SIOT, CMD, FAN and LAFTF1.1」의 난이도는 쉽다 본다

내가 Alkalic이 쉽다 생각하는 이유는 단순하지.

Alkalic을 방정식 ■나 잘하는 사람이 와서 푼다 가정하자. 정수론 올림피아드정도 나간 학생한태, 태입이 뭔지 설교한 후에 가르치면, 논리를 대수로 바꿔줘서 고맙다 할껄?

근데 사실은 정수론 올림피아드 나간 사람들 특징은 방정식이나 함수를 ■나 쉽게 풀어낸다는거야, 그말은 즉슨, 우리는 단순히 추상적인 연산과정인 함수 f, g, h에 대해, f(x) = g(x)나 h(x) = const. 꼴의 방정식 문제를 내는것으로 연산의 계산에 대한 서술 (두 계산이 같거나, 어떤 계산이 상수화됨)으로 방정식을 표현할수 있고, 이는 방정식이 본질적으로 함수(연산)에 대한 서술임를 나타냄, 그래서 이 FAN의 빈칸인 인자를, 단순히 연산해주는걸 가르쳐 주는거라서, 초등학생한태도 잘 설명할수 있는 중학고 수준으로 설명할수 있다는거지.

끝, 쉽지.

목표

- 「Alkalic Geogebra Sciance and Formal Explation of IT」이라는 책을 언젠가는 만들어서, 수학, 과학, 형식과학, 자연과학, 사화과학까지 다 Alkalic으로 서술 후 수학을 다시 formal하게 텍스트로 풀어낼것임.
- 크기개념같은 오만한 자기가 설명될거라고 믿는 합리화를 깨부수고, 그 개념이 더 humble하게, “나는 이렇게 생각합니다”로 말하게 하는것

최종 목표

- 언어 사상을 언어 재능이 결핍된 이들이나 속뜻 해석 능력이 결핍된 이들도 이해할수 있게 한다.
- 재능을 당연시하는 기존 방식보다는 기본적 사고 재능이 떨어지는 인간이 배울수 있는 방식을 쓴다
- 근거없는 멸시; 무재능자에 대한 멸시; 이•문과의 근거없는 비합리적 배타를 구축해 없엔다.

PART 6. 개연성

글의 구조와 개연성

PART1. 수에 대한 고찰

PART 2. 근본 있는 함수 인자 기반 연산의 활용 수학 체계 제안 (방정식 유사; 마인드는 연산의 서술)

PART 3. 방정식이 쉽다는 근거

PART 4. PROGRAMMING FW

PART 5. 그래서 뭐

태도는 사실상 5번에 다 정리되어있다.

3번은 기존 수학보다 2번이 더 어렵지 않냐는 바보같은 질문에 대한 반박인 샘이다.

수에 대해 고찰하여, 언어와 수의 권위를 깨부순다.

그 다음, 좀 더 받아들일수 있고 납득할수 있는, 배우는 자가, 해당 학문이 도구 형식 언어 수학에 대해 납득할수 있도록 만든, 겸손하고 재대로된 수학을 제안한다. (칸토어는 겸손하지 않다.)

그리고 마지막으로, 수학적인 프로그래밍 언어와 그 학습법을 제안하며, 마친다, 실제론 Edgar라고 안그래도 Unbeauty도 툴 언어인데, Edgar는 심지어 블록 코딩이다.

PART 7. 좌파적이지 않은 이유 feat. 우갈망 좌

우 갈망 좌 선언문은 다소 정치적이니
걸러듣으세요, 우파적 시각의 온건한 의견 출
제안문이빈다

우-갈망 좌파론

서론

나는 우파들이 우파적인 방법으로 우파적 목표를 이룰수 없을때 다소 좌파적인 수단을 동원하는것은 다소 극단적이라 본다.

그러나, 극단적이더라도, 우파의 최종 목표는 우파적 목표다.

따라서, 우파적 목표를 이루는것이 우파적 행동이 가야하는 바이다.

본론

1. 우파적 목표란 무엇인가? 우파적 목표는 보수적 정상상태이다.

우리는 보수적으로 정상화된 사회를 원한다. 그것이 우리의 이상향이기 때문이다.

그런데 보수적 정상상태는 세상이 부조리하기에 언젠가는 깨질수밖에 없다.

만약 필연으로 비정상화되는 보수 대재앙이 올때, 보수로써, 그런 상황을 막아야 할 것이다.

대(대적)-대재앙 상태이다.

그러나, 에초에 대-대재앙을 막기위해, 항시 평안함을 목표로 두고, 현실적으로 방어장치로써, 대재앙과 평안 깨짐을 막으려 들게 되어있다면, 나는 그것이 이상적인 사회라 본다.

이상에서, **평시 정상화**해야 한다.

평시 정상화가 답이다.

2. 평시 정상화되기 위해선 보수와 개혁보수가 공존해야한다. 정상화를 온건이 하거나, 개혁으로 정상화만 하면 안된다, 공존하지 않는것은 평안 깨짐을 만든다. 그러나 온건하고 평시 정상화를 이룰수 있는 방식으로 개혁하는것이 개혁이 평시 정상화를 깰 리스크가 있어도 할 이유가 되게 하는데 무엇이 문제일까?

개혁의 반복이나 극우적 행동이 목적일때, 보수적 색체인 평안에서 벗어나게 만든다. 얼마나 좌익적인가?, 그렇다, 너무 과한것은 그것이다, 개혁의 반복이나 극우적 행동이 목적이 될때, 그것은 보수가 아닌 좌익적 행동이나 다름없는 대제앙을 수반하는 행위인것이다, 우리는 대재앙을 막아야만 하는데 말이다!!!

3. 교활한 극좌들은 항상 예의주시해야한다. 그들은 좌익적 근거인 이상적 정상상태를 추구한다. 이상적 정상상태를 추구하는 그 사살과 행동, 실용적 실천을 위한, 우경적 수단과, 그러한 Agressival Movement까지, 그들의 행동을 예의주시하며, 그들이 하려는 대재앙을 막으려 항시 예방/대응해야 한다. 대응하는 방법은 파악한후에 백신을 만들듯, 더러운 재앙을 구축하는것이다. 책을 읽는 단계가 먼저다. 그들의 생각을 양이되어 처음엔 믿고, 두번째로 사자가 되어, 반항하라, 새번째로 어린이가 되어 타협점을 찾으라, 독서 다음도 당연히 알수 있다. 네번째는 독서된것을 보수적인 목적에 맞는지 검토한다, 그리고 다섯번째, 그들을 반박하고, 예측하고, 막고, 평안을 수호해야한다. 그것이 교활한 좌익들을 누구나 예측할수 있도록 해주는 기본적인 아이디어다.

이러한 방법으로도 안통하면 다른 방법을 찾아봐야 하지만, 그정도로 교활한진 아직 모르겠으니 넘기겠다. (예시로 빨갱이놈들 책이라고 공부하고 대처하여 더러운 사상을 새상에서 구축해 없애는것이다. 보수의 지성 엘리트적 특성, 인간의 지성의 역할이다.)

수단과 방법을 가리지 않고, 평안이 최고다.

"따라서".

이상적인 새계가 보수적 정상상태인 좌익을 생각하면,

이것이 우-갈망(Right-Desire) 좌파다.

위험한데;;

우-갈망 좌파적인 색체는 보수적인 잣대를 지키기 위해, 보수적 정상상태와 그의 과격한 일종인, 개혁을 마다치 않는 평시 정상화를 이상적인 새계로 잡고, 마다치 않는 평시 정상화를 이상적인 새계로 잡고, 옳은 "우"를 갈망하여 행동하는것이 목표인 좌파다.

대안(Alternative)우파로는 부족하다!!!

우-갈망을 부르짖어야 한다!!!!!

저 미친 극우친구의 말을 걸러 듣도록 하자
저놈 정치 싫어한다

정상상태 지향

이 교육 체계가 지향하는 바이다

이 교육은 정상상태를 지향하고 가치 중립적인 체계라 진보 진영이랑 보수 진영이랑 무관하다.