한국수학사학회지 제17권 제4호(2004년 11월), 45-64

λ-연산 소개

덕성여자대학교 불어불문학과 **정계섭**
kscheong@duksung.ac.kr

λ-연산은 '다시쓰기 규칙'으로 정의되는 계산을 위해 함수들이 형성되고, 결합되고, 활용되는 수학적 형식 체계이다. 컴퓨터과학의 발전과 더불어 많은 프로그래밍 언어들이 λ-연산을 원리로 삼고 있다. 나아가서, '커리-하워드 대응' 덕분에 이제 연역에 의해 수행된 증명과 컴퓨터 프로그래밍 사이에 대응 관계를 설정할 수 있게 되었다. 이 글의 목적은 교육적인 차원에서 아직은 잘 알려져 있지 않은 주제를 대중화시키는 데에 있다. 논리학과 컴퓨터 과학에서 λ-연산의 영향은 차후의 연구과제로 남아 있다.

주제어 : 커리화된 함수, 자유·속박 변항, λ-추출, 환원, 대체, 처치-로써 성질, 환원전략, 회귀, 커리-하워드 대응

0. 들어가는 말: 역사적 배경

칸토어(Cantor)는 무한의 문제를 수학적으로 해결하기 위하여 집합론을 구성하였는데 부랄리-포르티(Burali-Forti)의 역설(1897), 칸토어의 역설(1899), 러셀(Russell)의 역설(1903) 등이 발견되면서 집합론에 근거한 수학의 토대가 위협받는 상황에 이르게 되었다.

이에 독일의 수학자 힐베르트(Hibert)는 유클리드 기하학의 공리화보다 진일보한 수학의 형식적 공리화 작업을 추구하면서, 주어진 형식체계에서 임의의 적형식(well formed formula)이 이 체계의 정리인지 아닌지를 결정하는 알고리즘, 즉 기계적 절차가 존재하는지에 대한 문제를 탐구하였다.

30여년이 지난 1931년 괴델(Gödel)은 그러한 알고리즘은 존재하지 않는다는 불완전성 정리를 발표하여 학계에 큰 충격을 안겨주었다. 이 정리에 의하면 산수를 포함할 수 있을 정도의 임의의 무모순인 형식체계에서는 그 체계 내에서는 증명할 수 없는 참인 명제들이 존재한다는 것이다.

뒤를 이어 여러 가지 계산 모델들이 제안되었는데, 클리네(Kleene)의 부분순환함수(partial recursive function), 처치(Church)의 λ-연산(λ-calculus), 튜링(Turing)의 튜링 머신 등이 대표적인 것들이다.

원래 처치는 수학의 토대를 확립하기 위해 하나의 완전한 형식체계를 구축하려고 시도하였는데 λ-연산[1]은 이 형식체계에서 나온 함수에 관한 수학적 이론이다.

수학의 토대를 확립하기 위해 집합의 개념 대신에 함수의 개념을 원초적 개념으로 삼은 것은 1890년 프레게(Frege)가 처음인데, 쇤핀켈(Schönfinkel)의 1924년 논리체계[2]에서도 함수의 개념이 그 근저를 이루고 있다.

그러다가 처치의 논리체계에서 러셀의 역설과 유사한 역설이 발견되었는데 당시의 상황에서 이는 그리 놀랄 일이 아니다. 그래서 그는 그 체계에서 견고한 부분인 λ-연산을 '계산가능성'(Computability)을 연구하는 데 사용하였다. 그는 함수들($f: N^k \rightarrow N$)에 대한 계산가능성의 개념을 정의하기 위해 λ-정의가능성이란 개념을 도입하였는데, 양의 정수를 가진 함수의 함수값이 반복해서 대체(substitution)에 의해 계산될 수 있는 경우에 그런 함수를 λ-정의가능하다고 말한다.

저 유명한 '결정문제'(decision problem)에 대한 처치의 견해를 참고삼아 인용하겠다.

> By the *Entscheidungsproblem* of a system of symbolic logic is here understood the problem to find an effective method by which, given any expression Q in the notation of the system, it can be determined whether or not Q is provable in the system.[3]

1936년 그는 '결정불가능성 정리'(undecidability theorem)를 통하여, 유한 회의 단계를 거쳐 어떤 논리식을 증명하기 위한 결정절차가 존재하지 않음을 밝혔다. '결정문제'에 부정적 답변을 준 것이다.

이렇게 처치가 λ-연산을 통해 계산가능성의 개념을 탐구할 때, 튜링은 튜링머신에 기초한 계산모델을 가지고 같은 일을 하였다. 여기에서 '처치-튜링 논제(Church-Turing thesis)가 나오는데, 이에 의하면 튜링머신에 의해 계산가능한 함수는 λ-연산에 의해서도 계산가능하다. 이 논제는 아직 증명되지 않고 있는데, 그 이유는 계산가능한 함수와 계산불가능한 함수의 구분이 아직 분명하지 않기 때문이다. 그러나 추측(conjecture)이라 하지 않고 논제(thesis)라고 하는 이유는 이 주장이 옳을 것이라는 심증이 강하기 때문이다.

아무튼 1930년경 처치에 의해 발명된 λ-연산은 모든 계산가능한 함수를 표현할 수 있는 그리고 매우 단순한 통사론을 지닌 하나의 함수이다. 또한 계산가능성에 대한 간명하고 개념적인 모델이다. 앞으로 보게 될 두 가지 λ-어휘(또는 λ-표현) 형성규칙

1) 러셀과 화이트헤드는 함수 f(x)=2x+1을 2x̂+1로 고쳐서 표기했다. 처치도 처음에 x̂, 2x+1로 하려고 했으나 식자공이 ^x, 2x+1로 했고 또 다른 식자공이 λx, 2x+1로 바꾸었다고 한다.
2) 여기에서의 수학적 부분이 조합논리(combinatoric logic)인데, 이는 외면적으로 λ-연산과 동치임이 나중에 밝혀졌다.
3) Church, "A notre on the *Entscheidungsproblem*," *Journal of Symbolic Logic* 1(1936), pp. 40-41.

과 β-환원을 핵심으로 몇 개의 계산규칙을 가지고 모든 부분순환함수를 표현할 수 있는 강력한 기법이다.

더욱 획기적인 것은 λ-연산은 컴퓨터과학의 발전과 LISP, Haskell, Miranda, ML, CAML 등 함수 프로그래밍 언어의 등장과 더불어 다시 큰 각광을 받게 되었다는 사실이다.[4] 나아가서 그 이론적 중요성은 증명과 프로그램 사이에 '커리-하워드 대응'(Curry-Howard Correspondence)이 성립되어 훨씬 증대되었다.

1997년 프랑스의 수학자이자 논리학자 크리빈(Jean-Louis Krivine)은 λ-연산이 모든 수학적 추론과 수학적 구조를 표현할 수 있다고 주장하였다.

그에게 있어 λ-연산은 인간이 사고할 수 있는 가능한 모든 것의 '알파벳', 즉 라이프니츠(Leibniz)가 말한 의미에서 '보편언어'(caracteristica universalis)가 되는 셈이다.[5]

이 글에서 우리의 초점은 λ-연산 그 자체에 있다. 그 이유는 이토록 중요한 λ-연산이 과문의 탓이겠으나 국내에 비교적 잘 알려져 있지 않다고 생각하기 때문이다. 그래서 수많은 자료들을 섭렵하고 나서 처음 입문하는 사람들의 어려움을 조금이나마 해소하기 위해 우리 나름대로 λ-연산을 정리했음을 밝혀둔다.

1. 통사론

1-1. λ-표현들

함수기호 f(x)는 함수도 나타내지만 동시에 함수의 값을 표현하기 때문에 이중적인 의미를 지니는 불편함이 있다. 함수 자체를 표현하기 위해 도입된 장치가 바로 λ-연산자로서 다음과 같은 동치가 성립하는데, 좌우 모두 절차로서의 함수라는 점을 보여주는 점에서 모호성이 해소된다.

$$\lambda_x.f \equiv x \mapsto f(x)$$
$$\lambda_x.x^2 \equiv x \mapsto x^2$$

반면에 $\lambda_x.f$와 f(x)는 부분적으로만 동치일 뿐이다. 다시 한번 말하자면 f(x)는 결과에 도달하는 방식이나 절차보다는 결과 자체를 의미할 수 있기 때문이다.

일반적인 λ-표현은 다음과 같다.

$$\lambda_x.M\,(\cdots\ x\ \cdots x\ \cdots)$$

4) 함수언어란 함수들을 사용하는 프로그래밍언어를 말하는데 하나의 함수는 다른 함수들을 부르고 계속해서 그 함수들은 다시 또 다른 함수들을 부르고 하는 식이다.

5) 「과학과 생」, 1013호, 2002년 2월

M은 몸통(body), λ_x를 λ-추출자라 하는데 M 속에 있는 변항을 추출해서 $x \mapsto M$ ($\cdots x \cdots x \cdots$)라는 함수를 만든다는 것이다. "$x \mapsto M$"이 바로 λ_x이다. M 속의 자유변항 x는 λ_x에 의해 속박변항이 된다.

집합을 구성하는 {x | x is a boy}와 같은 표현도 λ_x.Boy(x)로 번역할 수 있는데, Boy(x)라는 개방명제를 만족시키는 개체를 추출한다고 해서 λ-추출(λ-abstraction)이라고 부른다.

λ-연산이 중요한 이유는 함수를 정태적으로 보는 것이 아니라 동태적으로, 즉 결과를 산출하기 위한 절차로서 파악한다는 점인데, 앞으로 보게 되겠지만 이는 프로그래밍 언어에서 매우 중요한 함축을 지닌다.

λ-표현들은 다음과 같이 정의된다.

① 모든 변수 : x, y, z, u, ⋯

② 기정의된 상수 : 수(numbers), 덧셈(addition), 곱셈(multiplication), 계승자함수 (successor function),[6] ⋯

③ 만일 X와 Y가 λ-표현이면, XY도 λ-표현이다.

④ x가 변항이고 M이 λ-표현이면 λ_x.M도 λ-표현이다.

이들 4가지 λ-표현들은 모두 동등한 자격이라는 사실을 유념해야 한다. 즉 모든 λ-연산의 표현은 다른 모든 λ-표현에 적용(application)될 수 있다. 이 말은 λ-연산에서는 정의역이나 치역이 없다는 의미이다.

1-2. 표기 관계

① 적용에서의 좌측 결합(left association)

다음의 동치가 성립한다.

$$M_1M_2M_3 \cdots M_k \equiv (((M_1M_2)M_3) \cdots M_k)$$

그래서 $(\lambda f.\lambda z.f(fx))(\lambda_x.x+1)3$과 같은 λ-표현에서 $M_2=(\lambda_x.x+1)$이고 $M_3=3$이며 나머지가 M_1이 된다.

일반적으로 (E_1E_2)에서 E_1은 함수로서 미리 정의된 상수거나 (계승자 함수 등), λ-추출 즉 (λ_x, M)로서 E_1을 E_2에 적용하라는 의미이다. 적용의 구체적 의미는 M 안에 있는 모든 자유변수 x를 E_2로 대체하고 나서 M을 산정하라는 것이다. 여기에서 E_1을 연산자(operator) E_2를 피연산자(operande)라고 한다. 어떤 임의의 두 λ-표현도 적용으로 간주될 수 있다. 예컨대, xx나 aa도 적용의 사례로 볼 수 있다는 말이다.

6) 수 이외의 다른 상수들은 응용 λ-연산에서 사용된다. 우리는 필요에 따라 이 상수들을 임의로 활용할 것이다.

② 추출에서의 오른쪽 결합(right association)

$\lambda_{x1x2} \cdots x_k.T$는 $\lambda_{x1}\lambda_{x2} \cdots \lambda_{xk}.T$의 생략된 형태이고 후자는 아래 식과 동치이다.

$$\lambda_{x1}\lambda_{x2} \cdots \lambda_{xk}.T \equiv (\lambda_{x1}(\lambda_{x2} \cdots (\lambda_{xk}.T)))$$

③ λ_x의 범위는 가능한 한 오른쪽으로 멀리 미친다.

$$\lambda_x.E_1E_2E_3 \equiv (\lambda_x.(E_1E_2E_3))$$[7]

여기에서 우리는 적용(application)이 추출(abstraction)보다 우선이라는 사실을 알 수 있다.

④ $(\lambda_x.M)a$와 같은 형태를 β-redex라고 하는데, “reducible”이라는 어휘에서 유래한다. 앞으로 자주 만나게 되는 형태이다.

다음과 같은 λ-표현을 이상의 규칙에 따라서 괄호가 생략되지 않은 형태로 만들어 보자.

$$\lambda_{xyz}.x(yz)t$$

i. 먼저 β-redex를 확인하라. 이를 위해 규칙 ②를 활용하면,

$$(\lambda_x \; (\lambda_y \; (\lambda_z \; (\; (x \; (yz) \;)t \;) \;) \;) \;)$$

① ② ③ ④⑤ ⑤④③②①

ii. 여기서 규칙 ③을 활용하면 λ_z, λ_y, λ_x의 몸통(body)이 드러난다.

iii. 남은 일은 x(yz)t에 규칙 ①을 활용하면 ((x(yz))t)가 되어 최종결과는 괄호 ⑤가 추가된 형태가 된 것이다.

연습을 위해 하나의 예를 더 추가하기 위해 다음 λ-표현을 살펴보도록 하자.

$$(\lambda n.\lambda f.\lambda x.f(nfx))(\lambda g.\lambda y.gy)$$

i. 먼저 β-redex를 확인한다. 이때 λ-연산자의 범위는 가능한 한 멀리 오른쪽으로 미친다.

(λx.f(nfx))	(λy.gy)
(λf.(λx.f(nfx)))	(λg.(λy.gy))
(λn.(λf.(λx.f(nfx))))	

ii. 다음 적용(application)의 왼쪽 결합 법칙을 활용하여 완전하게 괄호가 쳐진 표현을 얻게 된다.

$$(\; (\lambda n. \; (\lambda f. \; (\lambda x. \; (f \; (\; (nf)x \;) \;) \;) \;) \;) \qquad (\lambda g. \; (\lambda y. \; (gy) \;) \;)$$

⑤ ④ ③ ② ① ①②③④ ⑤

7) $(\lambda_x.E_1E_2)E_3$와 전혀 다른 λ-표현임을 유의할 것.

1-3. 커리화된 함수

λ-연산에서 모든 함수는 일항(unary)함수이다. 다음 두 함수를 비교해 보자.

① g(m, n) = n - m

② (λm.(λn.(sub nm)))

①에서 g는 물론 2자리 술어이다. 그러나 ②에 논항으로 3을 주면 결과는 한자리 술어 (n-3)으로서 다시 함수가 된다.

이를 유형(type)에 따른 정의역과 치역으로 나타내면, 다음과 같이 됨을 알 수 있다.

① : N × N → N　　　(N은 자연수)

② : N → (N → N)

즉, g는 한 쌍의 자연수를 취하여 하나의 자연수를 결과로 주는 반면에, λ-함수에서는 하나의 자연수를 취하여 그 결과로 자연수에서 자연수로 가는 함수를 준다는 것이다.

일반적으로, n개의 논항을 가진 함수 g는 하나의 논항을 취하여 (n-1)개의 논항을 가진 함수가 되고, 이 함수가 다시 하나의 논항을 취하여 (n-2)개의 논항을 가진 함수가 되고 이런 식으로 계속하여 마지막 단계에서는 하나의 논항을 가지는 함수가 되는 것이다.

$$f(n_1, n_2, n_3, \cdots, n_n) \rightarrow (((fn_1)n_2)n_3) \cdots$$

실상 $(\lambda_x.(\lambda_y.E))$는 $(\lambda_{xy}E)$를 커리화한 것이다. 그래서 일반적으로 $(\lambda_{x1}.(\lambda_{x2} \cdots (\lambda_{xk}.T)))$ '커리화된 함수'(curried function)라고 한다. 조금 다른 식으로 표현하자면 커리화한 $D_1 \times D_2 \times \cdots \times D_n \rightarrow D$와 같은 함수를 $D_1 \rightarrow (D_2 \rightarrow \cdots (D_n \rightarrow D\ ) \cdots)$ 처럼 본다는 것을 의미한다.

2. 의미론

2-1. 자유변항과 속박변항

λ-표현의 의미는 함수의 적용(application 또는 combination)이 전부 수행되고 난 후의 결과이다. 이것을 환원(reduction)이라고 하는데 이를 위해 자유변항과 속박변항 그리고 대체(substitution)의 개념을 먼저 검토해야 한다.

$(\lambda_x.M)$에서 λx는 본체 M 속에 있는 자유변항 x를 속박한다는 점을 이미 지적하였고, 그 외의 변항은 자유변항이다. 하나의 변항은 자유변항도 되고 속박변항도 될 수 있다.

$$(\lambda y.\ \underline{y}z)\ \underline{y}$$

속박변항 자유변항

$$\lambda x.\lambda y.(\lambda z.x\underline{y}z)y$$

자유변항

속박변항

'속박'의 의미는 λ-연산자가 직접 관할하는 본체에만 해당한다는 점을 유의해야 할 것이다.

E의 자유변항 FV(E)는 다음과 같다.

① 임의의 상수 C에 대하여 $FV(C) = \varnothing$

② $FV(x) = \{x\}$

③ $FV(E_1E_2) = FV(E_1) \cup FV(E_2)$

④ $FV(\lambda_x.E) = FV(E) - \{x\}$

예를 들면, $FV((\lambda_x.x)y) = \{y\}$이고, $FV((\lambda_x.x)x) = \{x\}$이다. ④의 경우, $FV(\lambda_x.xyz) = \{y, z\}$가 된다. $FV(E) = \varnothing$인 λ-표현을 닫힌(closed) 표현이라고 한다.

2-2. 대체

λ-연산에서 대체(substitution)의 개념은 매우 중요하다. 대체는 항상 λ-표현의 자유변항에 대해서만 이루어진다.

$$[x: = y]((\lambda z.zx)(\lambda x.x)$$
$$=([x: = y]\lambda z.zx)([x: = y]\lambda x.x)$$
$$=(\lambda z.zy)(\lambda x.x)$$

$(\lambda_x.x)$에서 x는 속박변항이기 때문에 [x:=y]가 아무런 영향을 끼치지 못함을 알 수 있다. 중요한 것은 대체의 결과 자유변수가 속박변항으로 되어서는 안된다는 사실인데, 대체규칙을 요약하면 다음과 같다.

① 임의의 변수 v를 E_1으로 대체하면 물론 E_1이 된다.

$v\ [v: = E_1] = E_1$

② v와 다른 변항 x에 대하여,

$x\ [v: = E_1] = x$

③ 임의의 상수 c에 대하여,

$c\ [v: = E_1] = c$

④ 연산자와 피연산자가 있을 때 교체는 각각에 적용된다.

(Erator Erand) [v: = E_1] = (E[v: = E_1]E[v: = E_1])

⑤ 교체는 λ-표현의 자유변항에만 적용된다.

(λ_v.E) [v: = E_1] = (λ_v.E)

⑥ x≠ v이고 x가 E1에서 자유변항으로 나타나지 않는다는 조건하에서 다음이 성립한다.

(λ_x.E) [v: = E_1] = λ_x.(E [v: = E_1])

⑦ 위의 경우 x≠v이고, x가 E_1에서 자유변항으로 나타나면 먼저 x를 새로운 변항 z로 바꿔주어야 한다.

λ_z. (E [x: = z] [v: = E_1])

이때 z≠v이고 또한 E나 E_1에서 자유변항으로 나타나서는 안 된다.

이상의 규칙들을 활용하여 다음의 경우를 살펴보자.

(λ_y.(λ_f.fx)y) [x: = fy]

여기에서 속박변항 y가 E_1 속에 자유변항으로 등장하기 때문에 ⑦에 의해 먼저 이 속박변항의 이름을 바꿔주면, 다음과 같이 된다.

λ_z.((λ_f.fx)z) [x: = fy]

그리고 ④에 의해 연산자와 피연산자에 대체가 각각 적용된다.

λ_z.((λ_f.fx) [x: = fy] z [x: = fy])

이제 ②에 의해, λ_z. ((λ_f.fx) [x: = fy]z)가 된다. 그런데 f∈Fv(fy)이므로 다시 ⑦에 의해, λ_z.(λ_g.(gx) [x: = fy])z가 된다. 이제 대체에 아무런 문제가 없으므로 시행을 하면 최종결과는 λ_z.(λ_g.g(fy))z가 된다.

3. 환원(Reduction)

3-1. α-재명명 (α-renaming)

속박변항의 이름은 중요하지 않다. 그래서 수학에서는 임시변수(dummy variable)이라고도 부른다.

$$(\lambda_x.x^2+1) = (\lambda_z.z^2+1)$$
$$\lambda_y.(x+y)(x-y) = (\lambda_z.(x+2)(x-2)$$

이렇게 속박변항의 이름을 바꿔주는 것을 α-재명명 또는 α-환원이라고 한다.

$$\lambda_{v}.E \xrightarrow[\alpha]{} \lambda_{w}.E\ [v: = w]$$

이때 w는 E에 나타나서는 안 된다. 아래의 예는 모두 α-환원이 올바로 적용된 경우들이다.

$$\lambda_{y}.(\lambda_{f}.fx)y \xrightarrow[\alpha]{} \lambda_{z}.(\lambda_{f}.fx)z$$

$$\lambda_{z}.(\lambda_{f}.fx)z \xrightarrow[\alpha]{} \lambda_{z}.(\lambda_{g}.gx)z$$

여기에서 주의해야 할 점이 있는데, 그것은 원래 자유변항이었던 것이 α-환원에 의해 속박변항이 되어서는 안 된다는 것이다. 이를 변항포획(variable capture)이라 한다.

$$\lambda y(x+y)(x-y) \not\xrightarrow[\alpha]{} \lambda x(x+x)(x-x)$$

$$\lambda x.\lambda y.xy \not\xrightarrow[\alpha]{} \lambda y.\lambda y.yy$$

변항포획을 피하기 위해서는 원래의 속박변항을 λ-환원에 의해 바꿔 주어야 한다. $[z: = x]\ (\lambda_{x}.zx)$와 같은 경우 원래의 자유변항 z가 대체결과 x가 되어 속박변항이 되므로 부당한 대체가 된다. 그래서 최초의 속박변항 x를 새이름 v로 바꿔주고 나서 대체를 시행하면 된다.

$$[z: = x]\ (\lambda v.zv) = \lambda v.xv$$

일반화해서 다음과 같은 λ-환원은 부당하다.

$$[z: = B[y]]\ (\lambda y.A[\cdots y\ \cdots z\ \cdots]) \not\xrightarrow[\alpha]{} (\lambda y.A\ [\cdots\ y\ \cdots B[y]\ \cdots])$$

여기에서도 최초의 속박변항 y를 u로 바꾸어주고 대체를 시행하면 안전하다. 앞에서 나온 대체규칙 ⑥, ⑦은 변항포획을 피하기 위한 규칙들이었다.

3-2. β-환원(β-reduction)

함수는 논항에 적용시키는 것을 β-환원이라 하는데, 본체(body)에 있는 모든 자유변항을 논항으로 대체시키는 것이다.

$$(\lambda v.M)a \xrightarrow[\beta]{} M[v: = a]$$

여기에서 좌변을 β-redex라 하고, 우변을 축약형(contractum)이라 한다.[8] β-환원이 수행되고 나서 더 이상 환원할 수 없을 때의 형태를 정상형태(normal form)이라고 한다. 다음은 β-환원의 사례들이다.

$$(\lambda_{x}.xy)z \xrightarrow[\beta]{} zy$$

8) 우변에서 좌변으로 진행하는 것을 β-추상(β-abstraction) 규칙이라고 한다.

$$(\lambda x.y)z \xrightarrow[\beta]{} y$$

$$(\lambda y.(\lambda z.a(yz)))bc \xrightarrow[\beta]{} (\lambda z.a(bz))c \xrightarrow[\beta]{} a(bc)$$

모든 λ-표현이 정상형태를 갖는 것은 아니다.

$$(\lambda_x.xxx)(\lambda_x.xxx) \longrightarrow (\lambda_x.xxx)(\lambda_x.xxx)(\lambda_x.xxx)$$

이는 프로그래밍 언어에서 반복실행명령을 할 때 중요한 의미를 지닌다.

경우에 따라 β-환원에 앞서 α-재명명을 해야 하는 필요가 생긴다. 다음은 논항이 함수인

경우 잘못된 환원의 예이다.

$$(\lambda_x.\lambda y.xy)\lambda z.\underset{\text{자유변항}}{\underline{y}}z \not\longrightarrow \lambda y(\lambda z.\underset{\text{속박변항}}{\underline{y}}zy)$$

그래서 속박변항 y를 v로 바꾸어주고 나서 β-환원을 수행해야 한다.

$$(\lambda_x.\lambda v.xv)\lambda z.yz \xrightarrow[\beta]{} \lambda v.(\lambda z.yzv)$$

λ-추출자가 여러 개 있을 수 있는데, 이런 경우 논항은 왼쪽으로부터 오른 쪽으로 가면서 쓰여진다.

$$\underset{4}{(}\ \underset{3}{(}\ \underset{2}{(}\ \lambda_x.\underset{1}{(}\ \lambda y.\underset{0}{(}\lambda z.x+y+z\ \underset{0}{)}\ \underset{1}{)}\ \underset{2}{)}\ 3\ \underset{3}{)}\ 5\ \underset{4}{)}\ 7$$

λ_x는 첫 번째 논항 3을 사용하고, λy는 두 번째 논항 5를, λz는 세 번째 논항 7을 사용해서 이 λ-표현은 3+5+7 = 15로 환원된다. 이로부터 적용(application)의 일반식을 도출할 수 있다.

$$\lambda(x_1,\ x_2,\ \cdots,\ x_n),M)N_1)N_2)N_3 \cdots N_n \xrightarrow[\beta]{}$$

$$M(x_1 := N_1,\ x_2 := N_2,\ \cdots,\ x_n := N_n)$$

충분한 수 이상의 논항이 있을 수 있는데, 이때 β-환원을 수행하고 남는 논항은 결과의 부분으로서 그대로 남겨두면 된다.

$$(\lambda_x.(\lambda y.yx))abcd \xrightarrow[\beta]{} bacd$$

3-3. η-환원과 δ-환원

본체 M 속에 자유변항으로 나타나지 않는 v가 λ-추출자가 되는 경우 이런 v는 무시하는 것이 η-환원이다.

$$\lambda v.(Mv) \xrightarrow[\eta]{} M$$

다만 M이 상수일 경우에는 η-환원이 적용되지 않는다.

$$\lambda v.(5v) \not\longrightarrow 5$$

그리고 M이 기정의된 함수일 경우에는 언제나 유효하다.

$$\lambda x.(sqrx) \xrightarrow[\eta]{} sqr$$

$$\lambda x.(add5x) \xrightarrow[\eta]{} (add5)$$

$$\lambda n.(mul2n) \xrightarrow[\eta]{} (mul2)$$

마지막으로 응용 λ-연산의 경우에 기정의된 함수를 적용시키는 것이 ∫-환원이다.

$$(add\ 3\ 5) \xrightarrow[\delta]{} 8$$

$$(succ\ 1\ 3) \xrightarrow[\delta]{} 14$$

$$(mul\ 4\ 7) \xrightarrow[\delta]{} 28$$

이제까지 4가지 기본환원방식을 점검해왔는데 대개 이 네 가지가 혼재되어 있는 경우가 흔하다. 예로서 β · ∫-환원을 보도록 하자.

$$(((\lambda f.(\lambda_x.(f(fx)))sqr)3)$$

$$\xrightarrow[\beta]{} ((\lambda_x.(sqr)\ (sqrx)))3$$

$$\xrightarrow[\delta]{} (sqr(sqr3))$$

$$\xrightarrow[\delta]{} (sqr9)$$

$$\xrightarrow[\delta]{} 8\ 1$$

4. 환원전략

4-1. 정상환원과 적용환원

모든 λ-표현이 정상형태로 환원될 수 없다는 사실을 우리는 이미 앞에서 보았다. 만일 정상형태가 존재한다면 이에 도달하는 방법은 한 가지 뿐인가? 이 질문에 대답하기 위해 다음과 같은 β-redex를 환원시켜보자.

$$x = (\lambda x.(\lambda y.(add\ y((\lambda z.(mul\ x\ z))3)))7)5$$

A. $$x \xrightarrow[\beta]{} (\lambda y.(add\ y)((\lambda z.(mul\ 7\ z))3))5$$

$$\xrightarrow[\beta]{} \text{(add 5((λz.(mul 7 z))3))}$$
$$\xrightarrow[\beta]{} \text{(add 5(mul 7 3))}$$
$$\xrightarrow[\beta]{} \text{(add 5 21)}$$
$$\xrightarrow[\delta]{} 26$$

그러나 똑같은 결과에 도달하는 다른 방법이 있다.

B. x $\xrightarrow[\beta]{}$ $(\lambda_x.(\lambda y.(\text{add } y(\text{mul } x\ 3)))7)5$
$$\xrightarrow[\beta]{} (\lambda y.(\text{add } y(\text{mul } 7\ 3)))5$$
$$\xrightarrow[\beta]{} (\lambda y.(\text{add } y\ 21))5$$
$$\xrightarrow[\beta]{} \text{(add 5 21)}$$
$$\xrightarrow[\beta]{} 26$$

A와 같은 방식을 정상순서환원(normal order reduction)이라고 하는데, 이는 바깥에서부터 시작하여 점점 안쪽으로 환원을 수행하는 방식이다.

이 방식은 결국 먼저 β-환원을 하고 나서 마지막으로 δ-환원을 수행하는 방식이라고 할 수 있겠다. 이에 반하여 B방식은 안쪽에서 시작하여 바깥쪽으로 환원을 수행하는 것인데, 말하자면 그때그때 가능한 경우에 δ-환원을 먼저 수행하는 방식이다. 즉 $(\lambda_x.Ma)b$와 같은 경우 본체 M 자신이 또한 함수이고 그에 대한 논항 a가 주어졌을 때 여기에서부터 환원을 시작하는 방식이다. 이런 환원을 적용순서환원(applicative order reduction)이라고 한다.[9]

정상환원과 적용환원 사이에 중요한 차이가 잇는데 그것은 정상형태가 존재하는 경우 정상환원은 이에 도달할 수 있는데 반하여 적용환원은 그렇지 못하다는 점이다.

A : $(\lambda y.5)((\lambda x.xx)(\lambda x.xx)) \longrightarrow 5$

B : $(\lambda y.5)((\lambda x.xx)(\lambda x.xx))$

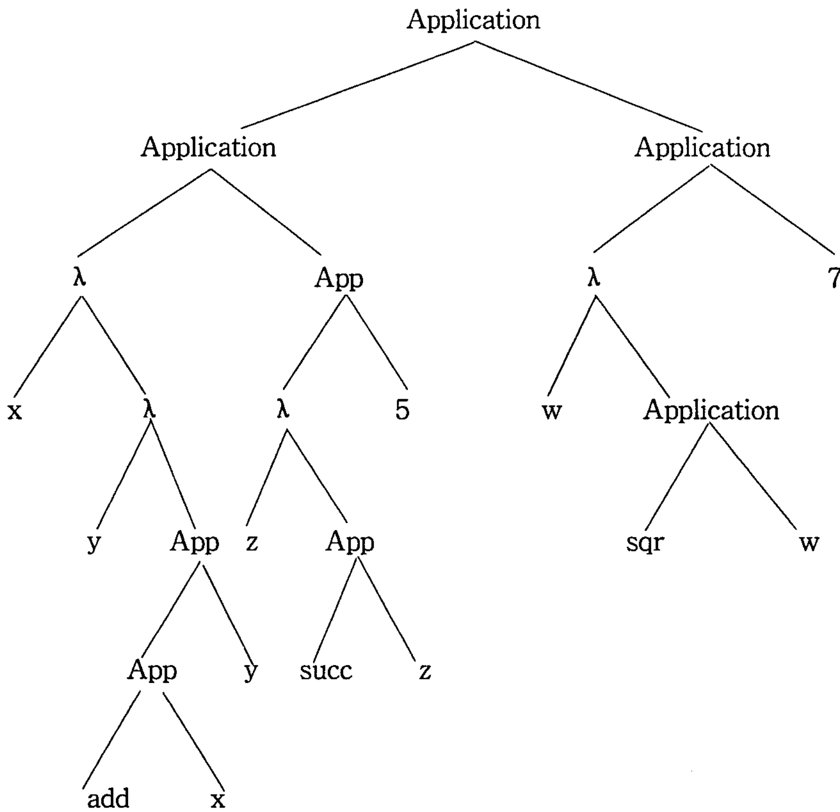
$\longrightarrow (\lambda y.5)((\lambda x.xx)(\lambda x.xx))$

$\longrightarrow \cdots$

어떤 경우에서건 β-redex를 신속하게 확인하는 것이 중요하다. $(\lambda v.M)a$와 같은 경우에는 E_1E_2가 분명히 드러나지만 대개의 경우 이 보다 훨씬 복잡한 표현들이 많다. 아래의 경우를 수형도로 분석해보자.

9) 프로그래밍 언어에서는 정상환원과 적용환원을 각각 'call-by-name' 그리고 'call-by-value' 라고 부른다.

$(((\lambda x.\lambda y.(add\ x\ y))((\lambda z.(succ z))5))((\lambda w.(sqr w))7))$



4-2. Church-Rosser 정리 Ⅰ, Ⅱ

앞에서 정상형태를 구하는 데에는 한 가지 이상의 환원방법이 있으며 각각 동일한 정상형태에 도달하는 것을 확인하였다. 이를 처치-로서 정리 1이라고 한다.

정리 1 : E, F, G가 λ-표현이고, E→F이고 E→G이면 F→H이고 G→H인 그런 H가 존재한다.

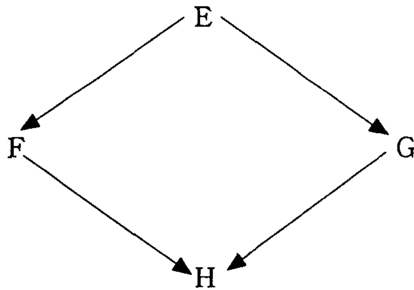
하나의 사례를 통하여 이 정리를 검토하도록 하자.

E : $(\lambda_x.(\lambda y.yx)z)v$

F : $(\lambda y.yv)z$

G : $(\lambda_x.zx)v$

H : zv



이런 성질은 다이어먼드 특성(diamond property) 또는 합류성(confluence property) 라고 한다.

앞에서 역시 만일 정상형태가 존재하는 경우 정상환원은 이에 도달할 수 있음을 보장한다는 사실을 보았는데 쳐치-로써 정리 2는 바로 이 사실을 의미한다. E가 λ-표현이고 N이 E의 정상형태일 경우 E에서 N으로 가는 정상순서환원이 반드시 존재한다는 것이다.

5 조합논리와 λ-연산

조합자(combinators)에 의해 구성된 함수와 λ-추출기에 의한 함수가 동치라는 사실은 이미 알려진 바이다.

I	≡	$\lambda_x.x$
C	≡	λxyz.xzy
W	≡	λxy.xyy
B	≡	λxyz.x(yz)
S	≡	λxyz.xz(yz)
T	≡	λxy.yx
∅	≡	λxyzw.x(yw)(zw)
Ψ	≡	λxyzw.x(yz)(yw)

여기에서는 하나의 사례를 통해 예시하는 것으로 그치겠다. Babc(=a(bc))가 λxyz.x(yz)abc와 동일한 결과를 주는지 검토해보자.

$$\text{Babc} = (\lambda_x(\lambda y(\lambda z.x(yz)))\text{abc}$$

$$\xrightarrow[\beta]{} \lambda y(\lambda z.a(yz))\text{bc}$$

$$\xrightarrow[\beta]{} (\lambda z.a(bz))c$$

$$\xrightarrow[\beta]{} a(bc)$$

유념해야 할 것은 여기에서의 동치는 어디까지나 외연적 동치일 뿐이라는 사실이다. 조합자들의 역할은 연산의 수행을 엄격하게 정해진 프로그램을 수행하는 데에 있다. 반면에 λ-연산은 λ-표현의 변항에 논항을 대체하는 연산(operation)에 기초한다. 다른 한편, 조합자에 의해 구성된 함수는 속박변항을 쓰지 않는다. 반면에 λ-연산에서는 대체(substitution)를 수행하기 위해 본체(body)에 있는 변항을 묶어 준다. 이런 점으로 보아 조합논리와 λ-연산은 내포적으로는 동치가 아니라고 결론을 내릴 수 있을 것이다.

6. λ-연산에 의한 프로그래밍 언어

λ-연산으로 컴퓨팅을 할 적에 필요한 구성자(constructors)로서 'Pair'와 선택자(selectors)로서 Head와 Tail은 아래와 같이 정의한다.

$$\text{Pair} \underset{df}{=\!=} \lambda a.\lambda b.\lambda f.fab$$

$$\text{Head} \underset{df}{=\!=} \lambda g.g(\lambda a.\lambda b.a)$$

$$\text{Tail} \underset{df}{=\!=} \lambda g.g(\lambda a.\lambda b.b)$$

이들 정의가 정확한지의 여부를 검산해보자.

$$\text{Tail (Pair pq)}$$

$$\Rightarrow (\lambda \mathbf{g}.g(\lambda a.\lambda b.b))$$

$$((\lambda \mathbf{a}.\lambda \mathbf{b}.\lambda \mathbf{f}.\mathbf{f}\ \mathbf{a}\ \mathbf{b})\ \mathbf{p}\ \mathbf{q})$$

$$\xrightarrow[\beta]{} ((\lambda \mathbf{a}.\lambda b.\lambda f.fab)\mathbf{pq})(\lambda a.\lambda b.b)$$

$$\xrightarrow[\beta]{} (\lambda \mathbf{b}.\lambda f.fpb)\mathbf{q})(\lambda a.\lambda b.b)$$

$$\xrightarrow[\beta]{} (\lambda \mathbf{f}.fpq)(\lambda \mathbf{a}.\lambda \mathbf{b}.\mathbf{b})$$

$$\xrightarrow[\beta]{} (\lambda \mathbf{a}.\lambda b.b)\mathbf{pq}$$

$$\xrightarrow[\beta]{} (\lambda \mathbf{b}.b)\mathbf{q} \Rightarrow\beta\ q$$

산수의 상수와 연산은 어떻게 정의되는가?

처치(Church)의 체계를 따르면, 자연수 n은 함수 f를 논항에 n번 적용하는 함수로 표현된다.

$$0 = \lambda f.\lambda_x.x$$
$$1 = \lambda f.\lambda_x.fx$$
$$2 = \lambda f.\lambda_x.f(fx)$$
$$3 = \lambda f.\lambda_x.f(f(fx))$$
$$\vdots$$
$$n = \lambda f.\lambda_x.(f(f(\cdots(fx)\cdots1) = \lambda f.\lambda_x(f^n x)$$

산수의 연산들은 다음과 같이 정의된다.

$$\text{succ} = \lambda m.(\lambda fx.(f(mfx)))$$
$$\text{add} = \lambda mn.(\lambda fx.((mf)(nfx)))$$ [10]
$$\text{mult} = \lambda mn.(\lambda fx.((m(nf))x))$$

여기에서도 정의의 정확성을 검토하기 위하여 한 가지 사례를 보도록 하자.

$$\text{succ1} \equiv (\lambda m.(\lambda fx.(f(mfx))))1$$
$$\xrightarrow[\beta]{} (\lambda m.(\lambda fx.f(mfx))))(\lambda x\lambda y.(xy))$$
$$\xrightarrow[\beta]{} \lambda fx.(f((\lambda x\lambda y.(xy))fx))$$
$$\xrightarrow[\beta]{} \lambda fx.(f(fx)) \equiv 2$$

Add 2 2

$$\Rightarrow (\lambda m.\lambda n.\lambda f.\lambda x.mf(nfx))$$
$$(\lambda g.\lambda y.g(gy))(\lambda h.\lambda z.h(hz))$$
$$\Rightarrow (\lambda n.\lambda f.\lambda x.(\lambda g.\lambda y.g(gy))f(nfx))$$
$$(\lambda h.\lambda z.h(hz))$$
$$\Rightarrow \lambda f.\lambda x.(\lambda g.\lambda y.g(\text{효}))f$$
$$((\lambda h.\lambda z.h(hz))fx)$$
$$\Rightarrow \lambda f.\lambda x.(\lambda y.f(fy))((\lambda h.\lambda z.h(hz))fx)$$
$$\Rightarrow \lambda f.\lambda x.(f(f((\lambda h.\lambda z.h(hz))fx)))$$
$$\Rightarrow \lambda f.\lambda x.(f(f((\lambda z.f(fz))x)))$$
$$\Rightarrow \lambda f.\lambda x.(f(f(f(fx)))) = 4$$

$$\text{mult12} \equiv (\lambda mn.(\lambda fx.((m(nf))x)))(\lambda x\lambda y.(xy))(\lambda x\lambda y.x(xy))$$
$$\xrightarrow[\beta]{} (\lambda fx.(((\lambda x\lambda y.(xy))((\lambda x\lambda y.x(xy))f))x))$$
$$\xrightarrow[\beta]{} (\lambda fx.(((\lambda x\lambda y.(xy))(\lambda y.f(fy))))x)$$

10) 이는 $\lambda m.\lambda n.(\lambda f.\lambda x((mf)(nfx)))$와 같다.

$$\xrightarrow[\beta]{} (\lambda fx.(\lambda y.(\lambda y.(f(fy)))y)x)$$

$$\xrightarrow[\beta]{} (\lambda fx.(\lambda y.(f(fy)))x)$$

$$\xrightarrow[\beta]{} (\lambda fx.f(fx))) \equiv 2$$

$$(\text{IS-ZERO})0 = (\lambda m.((m(\text{true false})\text{true}))(\lambda x\lambda y.y)$$

$$\xrightarrow[\beta]{} ((\lambda x\lambda y.y)(\text{true false}))\text{true}$$

$$\xrightarrow[\beta]{} \text{true}$$

나머지 함수들은 다음과 같은 true와 false의 정의에서 출발한다.

true = λxλy.x
false = λxλy.y

LISP와 같은 프로그래밍 언어는 λ-연산의 직접적인 임플러먼테이션 (implementation)[11]으로 볼 수 있다.

① let - expression
let x = 5 in (add x 3) ≡ (λx.(addx3))5

② where - expression
(add x 3) where x = 5 ≡ (λx.(add x 3))5

7. 회귀

회귀(recursion)의 개념에 들어가기 전에 먼저 고정점(fixed point)에 대해 알아보자. △가 임의의 집합이고 f가 △에서 △로 가는 함수라고 할 때, x = f(x)가 되는 △의 어떤 요소 x를 f의 고정점이라고 한다. f가 $(x^2+5)/6$이라고 할 때, 1과 5는 f의 고정점이 된다. f(x) = x^3의 경우 x = 1일 때 f(x)도 1이 되기 때문에 f의 고정점은 1이다.

어떤 함수가 주어졌을 때 그 결과로 새로운 함수를 주는 다음과 같은 함수를 고려해 보자.

R = λg.(λm.(λm.IF(IS-ZERO m)n(λq.q(PREDm)(succ n)))
= λm.(λn.IF(IS-ZERO m)n((PREDm)(succ n)))

지금으로서는 이 함수가 특별한 의미를 지니지는 않지만 어쨌든 이것은 하나의 함

11) 어떤 컴퓨터 언어를 특정 기종의 컴퓨터에 적합하게 함.

수이다.

이제 덧셈 연산을 다음과 같이 정의하자.

$$ADD = \lambda m.(\lambda n.IF(IS\text{-}ZERO\ m)n(ADD(PREDm)(succ\ n)))$$

위에서 본 R함수에 ADD함수를 적용하면.

$$RADD = \lambda m.(\lambda nIF(IS\text{-}ZERO\ m)n(ADD(PREDm)(succ\ n)))$$

따라서 ADD = RADD가 되고, 이는 ADD가 바로 **R**의 고정점이라는 사실을 말해주는 것이다. 즉 **R**의 고정점을 찾을 수 있다면 그것은 다름 아닌 ADD함수가 된다는 말이다.

애초에 우리는 **R**과 같은 난해한 함수의 고정점을 찾을 만한 아무런 단서도 없었다. 여기에 핵심이 있는데, λ-연산에서는 어떠한 함수에 대해서도 그 함수의 고정점을 찾을 수 있는 방법이 있다는 것이다.

사실 그러한 함수가 존재하는 데 이는 바로 Y함수로서 어떤 임의의 함수 f에 대해 Yf는 f의 고정점이 된다.

$$Yf = f(Yf)$$

그러한 Y가 존재한다는 것이 처음부터 명백한 것은 아니다. 이 Y는 다음과 같은 함수이다.

$$Y = \lambda f.((\lambda_x.f(xx))(\lambda_x.f(xx)))$$

Y를 임의의 f에 적용하면 다음과 같은 결과를 얻는다.

$$\begin{aligned} Yf &= \lambda f.((\lambda_x.f(xx))(\lambda_x.f(xx)))f \\ &= (\lambda_x.f(xx))(\lambda_x.f(xx)) \\ &= f((\lambda_x.f(xx))(\lambda_x.f(xx))) \\ &= f(Yf) \end{aligned}$$

바로 이것이 우리가 원하는 바인데, Y는 임의의 어떤 함수에 대해서도 고정점을 찾게 해주는 함수인 것이다. ADD함수가 **R**의 고정점인 것을 앞에서 보았으므로 ADD = Y**R**이 성립한다.

$$\begin{aligned} &(ADD\ one\ one) = ((Y\mathbf{R})\ one\ one) \\ &= (((\lambda_x.\mathbf{R}(xx))(\lambda_x.\mathbf{R}(xx)))\ one\ one) \\ &= ((\mathbf{R}(\lambda_x.(\mathbf{R}(xx))\lambda_x.(\mathbf{R}(xx))))\ one\ one) \end{aligned}$$

이 식을 풀어나가는 것은 매우 번거로운 일이지만 결국 결과는 2를 얻게 된다.

8. 나오는 말

λ-연산은 1930년 처치(Church)가 개발한 논리체계로부터 나온 함수에 관한 수학적 이론이다. 그 후 컴퓨터과학의 발전과 LISP, CAML과 같은 함수 프로그래밍 언어의 등장과 더불어 새로운 주목을 끌게 되었는데, 그 이유는 모든 λ-표현은 컴퓨터의 프로그램으로 간주될 수 있기 때문이다.

더욱이 1950년대 커리(Curry)와 하워드(Howard)에 의한 근본적인 발견으로 λ-연산은 결정적인 중요성을 뛰게 되었다. 흔히 '커리·하워드 대응'이라고 불리는 이 발견의 내용은, 연역규칙에 의해 수행된 각각의 증명에 대해 컴퓨터 프로그램을 대응시킬 수 잇다는 것이다,

수학에서 증명은 공리들로부터 출발하여 일련의 연역규칙을 적용하는 것으로 이루어진다. 다른 한편, 프로그램이란 컴퓨터의 프로세서에 보내는 일련의 명령들이다. 만일 하나의 명령을 각각의 연역규칙에 대응시킬 수 있다면, 수학에서의 증명은 일련의 명령 즉 프로그램으로 바꿀 수 있을 것이다.

예를 들어 modus ponens에 대응하는 명령은 λ-표현 적용(application)에 해당한다. 그리고 M이 <n, p>유형의 λ-표현이고, N이 n유형의 λ-표현이라면, (MN)은 p유형의 λ-표현이다. 여기에서 유형(type)은 적형식(fomula)에 해당한다.

<u>Modus Ponens</u>(e→)	<u>Application</u>
x → y	λx.sing(x) : n → p
x	Jean : n
y	(λx.sing(x))Jean : p

그리고 연역정리에 대응하는 명령은 λ-abstraction이 된다.

<u>Introduction →</u>	<u>λ-abstraction</u>
x	x : u
⋮	⋮
y	A(x) : v
x → y	λx.A(x) : u → v

즉, x가 유형 u이고, A(x)가 v유형의 λ-표현일 때, λx.A(x)는 <u, v>유형의 λ-표현이다.

이제 커리·하워드 대응이 함축하는 바를 충분히 이해할 수 있을 것이다. 이러한 사실에 고무되어 프랑스의 수학자 크리빈(Krivine)은 수학을 포함한 모든 언어는 뇌가 λ-연산을 활용하여 구축한 것이라는 매우 흥미로운 가설을 주장한 바 있다.[12] 앞으로 국내에서도 이 분야의 연구가 활성화되기를 기대한다.

참고 서적

1. H. Barendregt, *The lambda calculus*, North Holland, 1984.
2. J-Y. Girard · Y. Lafont · P. Taylor, *Proofs and types*, Cambridge University Press, 1989.
3. J. Hindley · J. Seldin, *Introduction to combinators and λ-calculus*, Cambridge University Press, 1986.
4. J-L. Krivine, *Lambda-calcul*, types et modéles, Masson, 1990.
5. Lalemend R., *Logique, réduction, résolution*, Masson, 1990
6. P. Martin-Löf, "Constructive mathematics and computer programming," In *Logic, Methodology and philosophy of science VI*, North Holland, 1979.
7. Renaud Francis, *Sémantique du temps et lambda-calcul*, PUF, 1996

λ-calculus

Dept. of French Language & Literature, Duksung Women's Univ. **Kye-Seop Cheong**

The lambda calculus is a mathematical formalism in which functions can be formed, combined and used for computation that is defined as rewriting rules. With the development of the computer science, many programming languages have been based on the lambda calculus (LISP, CAML, MIRANDA) which provides simple and clear views of computation. Furthermore, thanks to the "Curry-Howard correspondence", it is possible to establish correspondence between proofs and computer programming. The purpose of this article is to make available, for didactic purposes, a subject matter that is not well-known to the general public. The impact of the lambda calculus in logic and computer science still remains as an area of further investigation.

Key words: curried function, free and bound variable, λ-abstraction, reduction, substitution, Church-Rosser property, reduction strategies, recursion, Curry-Howard correspondence

2000 Mathematics Subject Classification : 03-XX, ZDM Classification : E4

12) 「과학과 생」, n⁰ 1013, 2002.2, pp.50-53