2008학년도 가을학기
SNU 4190.310 Programming Language
Homework 3 논술 과제

법의 계수 과정과 연관 지어 살펴 본 기계 중심 언어와 값 중심 언어의 발달 과정 및 구성체 운영 원리로서의 프로그래밍 언어 - C와 ML을 중심으로 하여

담당교수 : 이광근 교수님
해당교과목 : 4190.310 프로그래밍언어
제출일 : 2008. 10. 9.

법과대학 법학부 2008-12564 김재원

법의 계수 과정과 연관 지어 살펴 본 기계 중심 언어와 값 중심 언어의 발달 과정 및 구성체 운영 원리로서의 프로그래밍 언어 - C와 ML을 중심으로 하여

법학부 2008-12564 김재원

1. 들어가면서

사람은 저마다 살면서 '안경'을 하나씩 쓰고 살아간다. 인간은 저마다 세상을 보는 세계관, 인식관, 가치관 등이 있기 마련인데, 여기서 말하는 '안경'은 바로 그러한 것들을 통칭하는 것이다. 전문적인 용어로 패러다임이라고 칭하는 이 '안경'은, 일생을 살면서 평생 동안 그 정체가 조금씩 형성되고, 동시에 평생 동안 그 정체가 한 인간의 삶에 영향을 미친다. 자신의 '안경'을 통해 세상을 바라보는 것이 때로는 위험할 수도 있으나, 가끔은 신기한 통찰을 가져다주기도 한다고 생각한다.

이 글을 쓰고 있는 필자는 본래 법학도이다. 대학교에 입학하기 전 약 5년 정도 컴퓨터 공부를 한 인연이 닿아 지금 컴퓨터공학 공부를 병행하고 있으나, 그 5년 정도의 세월을 제외하고는 인문사회계열의 학생으로, 법학도를 꿈꾸는 학생으로, 그리고 법학도로 살아왔다. 본인이 본인 스스로의 '안경'을 평가하는 것이 아직은 섣부른 일일 수 없으나, 거칠게 평가해보자면, 필자의 '안경'은 보편적으로 공학도들이 가지는 그것보다는 법학도들이 보편적으로 갖는 그것에 더 가깝다고 평해볼 만하다.

법학도로서, 보편적으로 택하지는 않는 공학 공부를 병행하고 결심하게 된 것에는, 법학을 공학도의 '안경'을 통해, 공학을 법학도의 '안경'을 통해 관찰 및 탐구함으로써, '통섭'의 경지에 근접해보자는 욕심이 주된 이유가 되었다. 그래서 필자는 항상 공학 공부를 하는 데 있어서 기존의 시각을 벗어난 새로운 시각, 특히 필자가 몸담고 있는 '법학', 혹은 보다 폭 넓게, 사회과학의 시각을 견지하고자 노력해왔다.

Dennis M. Ritchie의 *The Development of the C language* (편의상 이하 이 글을 "*CDev*"로 칭하겠음), John McCarthy의 *A Basis for a Mathematical Theory of Computation* (편의상 이하 이 글을 "MTC"로 칭하겠음), Philp Wadler의 *Proofs are Programs: 19th Century Logic and 21st Century Computing*, (편의상 이하 이 글을 "19L21C"로 칭하겠음), 그리고 "계산이란 무엇인가에 대한 아이디어들"이라는 문서 (편의상 이하 이 글을 "계산"이라 칭하겠음) 등 4개의 글을 읽으면서도, 줄곧 독특한 시각, 차별화된 시각에서 이 문서들을 접근하려 노력하였다. 공통적으로 프로그래밍 언어에 대해 다루고 있는 이 4개의 글을 읽으면서도,

과연 프로그래밍 언어를 고찰하는 데 있어서 기존의 전산학에서 프로그래밍 언어를 바라보는 관점과 다른 새로운 관점으로 무엇이 있을까, 무엇이 새로운 '안경'이 될 수 있을까 지속적으로 고민하였다.

필자는 프로그래밍 언어의 발전 역사를 다룬 위 4개의 글을 읽으면서, 법의 계수 과정을 떠올렸다. 프로그래밍 언어가 발전해온 과정이 법을 계수해나가며 법과 제도를 발전시켜 나간 과정과 너무나 흡사했던 것이다. 이러한 관점을 시발점 삼아, 필자는 프로그래밍 언어를 바라보는 데 있어서 기존의 관점과는 차별화하여 프로그래밍 언어를 '구성체를 운영하는 원리'로 바라보는 '안경'을 착용하기로 하였다. 즉, '사회'라는 구성체를 운용하는 원리로서의 법과 같이, '프로그램'이라는 구성체를 운용하는 원리로서의 '프로그래밍 언어'를 살펴보기로 했다는 것이다.

위와 같은 관점 및 의도를 가지고 이 글은 다음과 같이 구성된다. 이 글은 1) 네 글의 요지 및 접근 방법 분석을 통해 본 프로그래밍 언어의 발전 과정 개괄, 2) 법의 계수 과정 개괄 및 프로그래밍 언어의 발전 과정과 법의 계수 과정의 비교, 3) '구성체 운용 원리'로서의 법과 프로그래밍 언어의 의의, 의 세 부분으로 구성된다.

2. 프로그래밍 언어의 발전 과정 개괄 - 네 글의 요지 및 접근 방법 분석을 통해

프로그래밍 언어의 발전 과정을 요약해서 살펴보려면, 크게 기계 중심의 언어(imperative language)의 발전 과정과 값 중심의 언어(value-oriented language)의 발전 과정을 중심으로 살펴보면 충분할 것이다. 프로그래밍 언어의 발전 역사를 큰 틀에서 볼 때, 역사 초기에는 '기계 중심의 언어 (imperative language)'의 발전이, 최근 들어서는 '값 중심의 언어 (value-oriented language)'의 발전이 주요한 화두가 되고 있다고 볼 수 있기 때문이다.

우선 기계 중심의 언어의 역사를 살펴보기 위해, 기계 중심의 언어들 가운데 단연 선두주자라 할 수 있는 C의 발전 과정을 글 CDev를 통해 살펴보자. 기계 중심의 언어의 역사를 논하면서 C의 역사를 논하는 것은, C가 가장 먼저 폭넓게 보급되었고, 널리 쓰이게 되었으며, 무엇보다도 UNIX 시스템과 뗄래야 뗄 수 없는 언어이기 때문이다.

C의 발전 과정에서 가장 두드러지게 나타나는 특징은, 언어의 개보수 과정을 통해 프로그래밍 언어의 진화를 진행시켜왔다는 점과, 이 때 '호환성'을 굉장히 높은 가치에 두고 있다는 점이라 할 수 있다. 우선 C 언어가 개보수 과정을 통해 진화를 거듭해왔다는 부분을 살펴보자. 글 CDev에 의하면, C는 NB의 개보수 결과, NB는 B의 개보수 결과, B는 BCPL의 개보수 결과이다. 즉, C라는 언어가 어느 날 갑자기 만들어진 것이 아니라, 오래 전부터 존재한 언어들의 개보수를 통해 만들어졌다는 것이다. 이러한 발전 과정은 프로그래밍의 본질과 그와 관련된 개념들을 다루고

있는 글 “계산”과 연관지어 볼 때 프로그래밍 언어의 ‘구성체 운용 원리’로서의 특징을 잘 보여주는 것이라 할 수 있는데, 이와 관련해서는 3절에서 법의 계수 과정과 연관시켜 더 자세히 고찰하도록 하겠다.

다음으로 C가 호환성에 굉장히 높은 가치를 두고 발전해왔다는 부분을, 글 CDev가 서술하는 기존 언어들의 개보수 과정 속에서 C에 새로이 추가된 개념과 C가 구현을 포기한 개념들을 통해 고찰해보자. C가 구현을 포기한 개념으로는 typeless characteristic 등이 있다. 글 CDev에 의하면, 지금의 C와는 달리 예전의 언어들은 typeless language를 추구했으나, 지금의 C로 발전하면서 typeless languae의 구현을 포기했다고 한다. C가 새로이 갖게 된 개념으로는 부동 소수점 표현, 포인터 사용, struct 사용 및 포인터 지정 등이 있다.

생각건대, CDev가 묘사하고 있는 개보수 과정 가운데 가장 드라마틱한 것은 C가 기존의 typeless characteristic을 버리고 type를 채용하고, 포인터라는 개념을 도입한 부분이라고 할 수 있다. 타입의 복잡한 존재와 포인터의 존재, 그리고 typeless language인 시절의 프로그램들까지에 호환성을 제공하기 위한 다양한 표현상 별칭들의 존재 등은, C의 강력한 호환성과 폭넓은 적용 범위를 갖춘 지금의 C의 모습을 만든 일등공신들이지만, 또한 program analysis 분야에서 C가 맹렬히 비판받고 있는 이유들이기도 하기 때문이다. 이러한 드라마틱한 변화를 갖게 된 배경에 호환성을 추구했던 설계자들의 노력이 있었음을 우리는 간파해야 한다. 글 CDev에는 언어를 발전시켜나가면서 발생하는 문제에 대처함과 동시에 기존의 언어들로 작성된 프로그램과도 호환성을 갖출 수 있도록 안간힘을 쓴 설계자들의 노력이 잘 묻어나있다. 원시적인 언어로부터 완성된 현재 형태의 C로의 발전 과정은 어떻게 보면 ‘호환성’을 위한 ‘타협’의 과정이었다 해도 과언이 아닐 정도이다. CDev에 나타난 C를 설계한 프로그래머들의 고민은, 약간 과장해서 이야기하자면 ‘어떻게 하면 완전무결하고 깔끔한 언어를 만들까’ 보다는 ‘어떻게 하면 기존에 쓰이던 언어의 프로그램들도 돌려가면서 기존에 쓰이던 언어가 가진 문제점들을 해결할까’ 하는 것이라고 할 수 있다. 이러한 호환성 역시 구성체를 운영하는 원리로서 갖춰야 할 주요한 덕목 가운데 하나라는 점에서, 프로그래밍 언어의 구성체 운영 원리로서의 특성을 잘 보여주는 요소라 할 수 있다.

다음은 값 중심의 언어의 발전 과정을 살펴보자. 글 19L21C와 글 MTC는 프로그래밍 언어와 수학적 모델을 연관 지어 설명하고 있다. 글 19L21C는 computation이 수학적 모델링을 거친 수학적, 논리적 프로세싱과 별반 다르지 않다는 이야기를 하면서, ML 등 값 중심의 프로그래밍 언어들이 이러한 수학적 모델을 따라 만들어진 언어라는 점을 숨기지 않고 이야기한다. 글 MTC 역시 수학적, 논리적 모델을 바탕으로 computation에 사용된 여러 개념들을 설명하면서, universal language의 존재를 언급하는데, 이러한 언어가 값 중심의 언어라고 볼 수 있다.

글 19L21C와 글 MTC를 종합해볼 때, 값 중심의 언어는 다음과 같은 과정을 거쳐 착안되고 만들어졌으며 응용되었다고 할 수 있다. 기존의 기계 중심의 언어는

앞서 언급한 바와 같이 호환성이 높고 상대적으로 적용 범위가 넓으나, 상시적으로 오류가 발생할 가능성이 있고 static program analysis가 불가능한 등 여러 가지 문제 또한 있음이 시간이 지남에 따라 지적되었다. 프로그램의 완전무결성이 더욱 강조되고 있는 최첨단 시대에 이러한 결점은 치명적인 것이라 할 수 있다. 프로그래밍 언어의 설계자들은 이제 더 이상은 기존의 패러다임 하에서 '개보수'를 통해 문제를 해결하는 것이 불가능하다고 판단, 새로운 모델링을 시도한다. 그 때 그들의 눈에 들어온 것이 수학적, 논리학적 모델링인 것이다. Gentzen의 proof method와 type checking의 연결, Church의 lambda calculus 을 기반으로 한 functional programming의 가능성 개척 등은 이러한 모델링의 핵심을 이루고 있으며, 이를 바탕으로 만들어낸 언어가 ML, Scheme 등의 값 중심의 언어인 것이다.

정리해보건대, 값 중심 언어의 발전 과정에 있어서 가장 두드러진 특징은 수학적, 논리학적 모델링을 기반으로 했다는 점과 기존의 패러다임을 뒤엎었다는 점이라 할 수 있겠다. 우선 수학적, 논리학적 모델링을 기반으로 했다는 점에 대해 살펴보자. 글 19L21C에 의하면 프로그램은 결국 수학에서 아주 오랫동안 계속 연구해오고 실행해오던 증명의 과정을 컴퓨터에 입력한 것일 뿐이다. 따라서 프로그래밍에서 기본적으로 사용되는 개념들도 모두 수학적 개념들로 표현될 수 있다는 것이다. 글에 명시적으로 나와 있지는 않으나, 글 19L21C와 글 MTC 모두 암묵적으로 그러한 전산학 전반이 수학적 개념에 기반하고 있음에도 불구하고 기존의 언어들은 그러한 수학적 개념이 드러나지 않게 설계되었다는 점을 지적하는 논지를 견지하고 있다고 할 수 있다. 이는 생각건대 기존의 전산학자들이 프로그램의 존재를 정의내리고 발전시키는 데 있어서 수학과 전산학의 유사점을 발견하지 못한 데에서 기인했다고 할 수 있다. 이제는 그러한 유사점들을 점차적으로 발견 및 탐구하고 있으므로, 프로그래밍 언어가 수학적 모델을 차용하여 설계됨은 물론 그 겉모양까지 마치 수학의 증명을 작성하듯 보이게 설계되고 있다.

둘째로 기존의 패러다임을 뒤엎었다는 점을 살펴보자. 앞서 언급한 바와 같이 값 중심의 언어는 기계 중심의 언어가 가지고 있던 문제점을 실질적으로 해결하는 것이 기계 중심의 언어를 개보수함을 통해서는 불가능하다고 판단하여 만들어진 것이다. 즉, 기계 중심의 언어의 본질을 이루고 있는 기본적인 모델을 거의 전부 폐기하고 새롭게 만들어진 것이다. 앞서도 언급하였고 이 글에서 계속적으로 언급할 '구성체 운영 원리'로서의 프로그래밍 언어의 의의를 기반으로 생각하여 볼 때, '혁명'에 비견해볼 수 있겠다.

이상으로 기계 중심 언어와 값 중심 언어 각각의 역사 및 의의를 살펴봄으로 해서 프로그래밍 언어 전반의 역사 및 의의를 조명해 보았다. 하지만 이렇게만 살펴보는 것은 기계 중심 언어와 값 중심 언어를 너무나도 동떨어진 것으로 보게 하기 쉽다. 따라서 우리는 또한 전산학이 근본적으로 해결해야 하는 문제, 혹은 전산학이 근본적으로 의문을 품고 있는 문제 및 그와 관련된 개념을 또한 확실히 정립할 필요가 있다. 이런 근원적인 부분들에 대해서는 글 "계산"이 잘 서술하고 있다. 글

“계산”은 semantic equivalence, sequentiality, interaction, mobility 등 네 가지 개념에 대해 서술하고 있는데, 이 개념들은 그 자체로 ‘개념’인 동시에 해결해야 하는 문제가 되는 것들이라 할 수 있다. 이러한 개념 또는 문제는 언어가 기계 중심 언어이든 값 중심 언어이든 간에 공통적으로 사용될 수, 혹은 적용 될 수 있다는 점에서, 두 종류의 언어들의 역사를 연관 지어 주고 연결시켜주는 역할을 하고 있다고 할 수 있다. 계속해서 언급하고 있는 구성체 운영 원리로서의 프로그래밍 언어의 의의라는 관점을 기반으로 하여 살펴보아도, ‘기계 중심’, ‘값 중심’이라는 것이 제도의 종류 및 특성을 특정하는 하위 개념이라면, 글 “계산”이 언급하는 개념 및 문제들은 그 구성체가 포괄적으로 가지고 있는 아주 기본적인 개념이나 내용, 정체성 등을 의미하는 것이라 할 수 있겠다.

이상으로 프로그래밍 언어의 발전 과정에 대한 검토를 마쳤다. 정리해보면, 프로그래밍 언어의 발전 과정은 기계 중심 언어의 발전 과정, 기계 중심 언어가 값 중심 언어로 변모해나가는 과정, 그리고 값 중심 언어의 발전 과정으로 세분화해서 살펴볼 수 있으며, 이 과정들 속에서도 전산학의 근본적인 물음들은 변하지 않고 계속해서 문제가 되고 있다.

3. 법의 계수 과정 개괄 및 프로그래밍 언어의 발전 과정과 법의 계수 과정의 비교[1)]

앞서 네 글들을 통해 프로그래밍 언어의 발전 과정을, 구성체 운영 원리로서의 프로그래밍 언어라는 대 전제적 관점과 연관 지어 살펴보았다. 그러면 이제 대표적인 구성체 운영 원리인 법이 어떻게 발전하는지 그 계수 과정에 대해서 간단히 살펴보고, 이러한 계수 과정과 프로그래밍 언어의 발전 과정을 연관 지어 생각해보도록 하겠다. 계수 및 발전 과정의 유사성, 계수 및 보수 원인의 유사성, 그리고 더 나아가 법과 프로그래밍 언어 자체의 유사성을 차례로 고찰하겠다.

법의 계수란 다른 국가나 민족의 법률제도를 수입하여 자기 나라의 제도로 채택하는 일을 말한다. 법의 발전은 대체적으로 기반이 되는 법 위에 다른 나라의 여러 법 제도들을 참고하는 방식으로 이루어진다. 어느 날 갑자기 법 제도를 하나 완전히 새로 만드는 것이 아니라는 것이다.

물론 계수의 원인으로 외부의 압력(정복, 지배 등)을 언급할 수도 있겠지만, 그보다도 계수는 법의 근원적인 특징이라고 보는 것이 더욱 타당하다. 그 이유는 법이라는 것이, 그리고 구성체의 운영 원리라는 것이 한 순간에 만들어낼 수 있는 단순한 것이 아니기 때문이다. 법은 모름지기 사회가 운영되는 원리를 모두 총괄하고 그에 대한 판단을 내릴 수 있어야 한다. 하지만 그러한 법이 규율해야 되는 이 사

1) 이 내용은 이 글의 논지 흐름 상 참고만 하면 되는 부분이므로, 기초적인 개념만 다루기 위해 백과사전의 내용을 기반으로 하여 작성하였음.

회는 너무나도 복잡다단하다. 법은 분명 이 사회에서 벌어질 수 있는 모든 일들을 예측하여 그 일들을 규율할 수 있어야 하는데, 이 사회의 예측불가능성은 법의 이러한 역할을 힘들게 한다. 따라서 법은 최대한 그 예측가능성을 높이려 노력하고, 그러기 위해 나온 방안이 바로 이 "계수"인 것이다. 한 순간 사람들 몇 명이 모여서 책상 앞에서 '앞으로 사회에 무슨 일이 일어날까'를 고민하는 것 보다는, 역사적으로 오랫동안 실험을 통해 검증된 사회 운영의 실험 결과들을 이용하여 법을 만드는 것이 더 효율적이라는 것이다. 우리나라 법 역시 계수의 예외가 아니다. 해방 이전까지의 전통법의 경우, 형법은 대체적으로 당률이나 대명률 등의 중국법이 계수된 것이라고 보는 것이 통설이고, 민법의 경우 우리나라의 자체적인 관습법을 기반으로 하고 있다고 보는 것이 통설이다. 그리고 해방 이후에는, 독일법이 계수된 일본법과 불문법의 전통을 계수한 영미법 특히 미국법이 혼합적으로 계수되어 현재의 법체계를 만들었다고 보는 것이 주된 입장이다.

그렇다면 프로그래밍 언어의 발전 과정과 법의 계수 과정에는 어떤 공통점이 있는지 살펴보자. 앞서 프로그래밍 언어의 발전 과정은 크게 기계 중심 언어의 발전 과정과 값 중심 언어의 발전 과정이라는 두 부분으로 나눠볼 수 있다고 언급하였는데, 법의 계수 과정과 큰 공통점을 보이는 부분은 두 부분 중 기계 중심 언어의 발전 과정이라 할 수 있다. 글 CDev에 의하면 Ritchie는 BCPL이 B로, B가 NB로, 그리고 C로 발전해나가는 과정 속에서 개발자들이 Algol 등 다른 언어의 개념들을 많이 참고했다고 고백하고 있다. 즉 완전히 새로 만들어진 개념이 아니라, 다른 언어가 비슷한 문제의식을 갖고 개발해 낸 비슷한 개념들을 이용한 것이라는 것이다.

하지만 값 중심의 언어까지 법이 계수되는 것과 같은 메커니즘으로 볼 것인가에 대해서는 논란이 있을 수 있다. 기계 중심 언어는 비교적 분명하게 기존 언어들을 고쳐가며, 다른 언어들에서 개념을 차용해가며 발전해온 것이 드러나지만, 값 중심의 언어는 그 계통 언어 간의 상호 관계도 잘 드러나지 않고, 그 계통 언어들의 발전 과정을 논할 만큼 그 역사가 오래되지도 않았기 때문이다. 값 중심 언어에 속한다고 볼 수 있는 SML, Ocaml, nML 등은 기본적으로 거의 유사하다고 판단함이 더 타당하고, ML 계열의 언어들과 scheme, LISP 등 역시 큰 틀에 있어서는 차이점보다 유사점이 더 많다. 즉, C가 발전함에 있어서 그 중간단계 역할을 했던 BCPL, B, NB 등의 언어와 '베낌의 피해자'가 되어주었던 Algol 등의 언어가 C와 다른 정도가, 값 중심 언어 군의 발전에 서로 영향을 끼친 언어들 간의 다른 정도보다 월등히 작다는 것이다. 하지만 서로 상호 작용을 하면서 서로 간의 개념을 차용하고 발전시켜 나가고 있다는 점에서는 어느 정도 '계수'의 메커니즘을 따르고 있지 않나 조심스럽게 이야기해볼 수 있다.

법의 계수의 주된 원인이 외부적 압력 보다는 법 자체의 근본적인 양태 및 취지 때문에 그렇다는 설명처럼, 프로그래밍 언어의 '계수' 원인 역시 그러한지에 대해 살펴보도록 하자. 결론부터 이야기하자면, 프로그래밍 언어의 계수 원인 역시 법의 계수 원인과 거의 유사하다고 할 수 있다. 프로그래밍 언어가 강력하다고 일컬어질

수 있는 이유는 프로그래밍 언어가 유한의 의미구조를 가지고 무한의 프로그램을 만들어낼 수 있다는 점에 있다. 이를 바꿔 이야기하자면, 법이 다루는 사회에 해당하는 프로그래밍 언어가 다루는 프로그램의 영역은 그만큼 무한히 넓다는 것이다. 그렇기 때문에 프로그래밍 언어 역시 그 언어를 사용하여 작성할 수 있는 프로그램의 종류가 무한하도록 모든 경우 가짓수를 다 반영할 수 있도록 설계되어야 한다. 하지만 앞서 법에 대해 언급한 바와 같이 사람 몇 명이 모여서 그렇게 만드는 것은 쉽지 않기 때문에, 오랫동안 여러 시행착오를 거쳐 오면서 여러 기능들이 보완된 다른 언어들을 참고하여 언어를 만드는 것이 타당성을 얻게 되는 것이다.

법과 프로그래밍 언어 자체가 유사성을 가지고 있음 역시 추론 가능하다. 필자는 C의 발전 과정을 앞서 살펴보면서 도출한 특징 중 하나인 호환성을 가치에 두는 개발 과정과 법의 정의 세부 관념 중 하나인 법적안정성의 유사성을 기반으로 하여 법과 프로그래밍 언어 자체의 유사성을 고찰하였다. 법을 만들고 운영하는 데 있어서 가장 기본적인 원칙이 되는 '정의(正義)' 개념을 세분화하여 살펴보면 정의관념, 합목적성, 법적안정성으로 나뉘는데, 이 중 법적안정성을 추구하는 것이 앞의 호환성을 추구하는 것과 그 원리 및 목적이 거의 비슷하다. 어떤 법이 정의롭다고 이야기하기 위해서는 그 법의 적용 자체가 정의 관념에 부합하는 것도 중요하지만, 그 법의 적용이 일관성이 있는 것도 굉장히 중요하다. 그러한 일관성에 기반을 두어 일반 시민들의 법적 생활이 유지되는 것이기 때문이다. 사회에서 벌어지는 케이스들을 법이 총괄하여 다룰 수 없다면, 시민들은 어떤 행동이 법적으로 옳고 어떤 행동이 법적으로 그름을 알 수 없게 되므로 안정적인 법 생활을 영위할 수 없게 된다. 프로그래밍 언어 역시 그러하다고 할 수 있다. 해결해야 하는 문제, 혹은 짜야 하는 프로그램들을 프로그래밍 언어가 총괄하여 구현할 수 없다면, 그 프로그래밍 언어는 존재 의의를 상실케 되는 것이다. 그렇기 때문에 프로그래밍 언어의 호환성이 중시되는 것이기도 하다.

이상으로 법의 계수 과정과 프로그래밍 언어의 발전 과정이 유사성을 띄고 있음을 살펴보았다. 이렇게 법의 계수 과정과 프로그래밍 언어의 발전 과정의 유사성, 법과 프로그래밍 언어의 유사성을 살펴보는 것은, 단순히 그 유사성을 밝히고 그치는 것이 아니라, 뒤의 절에서 다룰 '구성체 운영 원리로서의 프로그래밍 언어'를 설명하기 위해 의의가 있다고 할 수 있다.

4. '구성체 운용 원리'로서의 법과 프로그래밍 언어의 의의

앞서 2절에서 살펴본 프로그래밍 언어의 발전 역사를 살펴보고 필자가 가장 주요하게 느낀 점은 프로그래밍 언어의 실체를 어떻게 파악하느냐에 따라 그 발전 과정이 완전히 달라질 수 있다는 점이었다.

초기의 프로그래밍 언어 개발자들은 프로그래밍 언어를 '기계를 돌리기 위한 도

구' 정도로 밖에 안본 것으로 보인다. 그들에게 프로그래밍 언어란 단지 기계를 돌리는 데만 쓰면 그뿐인 것이므로, 그들이 만드는 프로그래밍 언어의 최종 목표는 결국 '기계를 잘 돌아가게 만드는 것'일 수 밖에 없는 것이다. 그들이 꿈을 작게 가진 것이라기보다는 그 당시에는 그 정도만도 상당히 달성하기 어려운 목표였을 것으로 생각된다. 여하튼 그들의 프로그래밍 언어에 대한 이러한 인식은, 프로그래머들에게 기계적 조작을 가능케 하는 등 그야말로 '기계를 가동키 위해' 필요한 최대한의 권한을 모두 부여했지만, 프로그래머들이 기계 조작을 넘어서 문제를 해결하게 하는 데에는 어려움을 가져다주었다. 문제를 해결하는 데에는 기계를 조작하는 선에서 그치는 것이 아니라, 그 기계를 조작하여 실제로 원하는 값을 도출케 해줘야 하기 때문이다. 하지만 기계 조작에 초점을 맞춘 그들이 개발한 언어는 기계 조작을 전적으로 가능케 하기 위해 프로그래머들에게 너무 많은 권한을 부여하여 문제의 실질적 해결까지 가는 데에 메모리 충돌, 타입 에러 등 수많은 사소한 문제들에 봉착하게 만들어 버렸다.

그에 비해 값 중심의 언어가 등장하기 시작하는 뒤 세대의 프로그래밍 언어 개발자들은 기계 중심 언어 개발자들의 사고를 넘어, '문제를 해결하기 위한 주요한 도구로서의 프로그래밍 언어'를 그들의 이상적 프로그래밍 언어 상으로 여긴 것으로 보인다. 그들의 프로그래밍 언어가 차용한 수학적·논리학적 모델은 수학자들이 수학 문제를 풀기 위해, 논리학자들이 논리학 문제를 풀기위해 '도구'로 볼 수 있는 것들인데, 그들이 프로그래밍 언어를 만들 때 이러한 요소들을 차용했다는 것은 프로그래밍 언어가 그들에게는 프로그램에게 주어지는 문제를 푸는 도구로 인식되었다는 점을 증명하는 것이라 생각한다. 프로그래머들의 기계 조작 권한을 조금 줄인 대신 문제 해결에 곧장 도달할 수 있도록 필요한 타입 체킹, 메모리 누수 체킹 등을 프로그래밍 언어가 대신 해줄 수 있게 프로그래밍 언어를 설계한 것이다.

이처럼 개발자들이 프로그래밍 언어의 궁극적 의미 및 목적을 어떻게 보느냐에 따라 그들이 개발하는 프로그래밍 언어의 양태는 엄청나게 달라졌다. 필자가 여기서 법과 프로그래밍 언어를 비교해 가면서 구성체 운영 원리로서의 프로그래밍 언어의 의의를 부각시키고자 하는 목적도 여기에 있다.

구성체 운영 원리는 기본적으로 그 원리에 의해 규율 받는 구성체 내부의 구성원 각각에 대한 고려가 중요하다. 그 구성원들이 상호 간에 어떻게 작용하고 영향을 주고받으며, 그 구성원들이 어떤 형태로 존재하고 소멸하는지에 대해 관심을 갖는 것이 운영 원리의 핵심을 이루고 있다고 볼 수 있다. 프로그래밍 언어에 이와 같은 개념을 적용시켜 보자. 구성체는 프로그램이 될 것이 자명하므로, 그 구성체의 구성원들은 함수, 변수, 등의 object를 가리키는 것이라고 봐도 무방하겠다. 이러한 오브젝트들이 상호 간에 어떻게 영향을 주고받는지, 이러한 오브젝트들이 어떤 형태로 어떤 프로시저를 거쳐 생겨나고 없어지는지에 대해 관심을 갖자는 것이다. 실제로 이러한 식의 접근을 객체 지향적 프로그래밍(object-oriented programming)에서 하고 있다. 또한 전산학에서 중요하다 할 수 있는 관계 중심의 사고법 역시 이

러한 ‘구성체 운영 원리’로서의 프로그래밍 언어에의 접근에 의해 뒷받침된다고 할 수 있다.

또한 ‘구성체 운영 원리’로서의 프로그래밍 언어에의 접근은 향후 프로그래밍 언어의 발전 과정에도 특정한 통찰을 줄 것이라 믿어 의심치 않는다. 구성체를 운영하는 원리가 갖춰야할 가장 중요한 덕목을 하나 뽑아야 한다면, 필자는 ‘안정성’을 뽑겠다. 그냥 ‘기계’ 하나를 돌리는 원리라면 그 기계 하나의 사정만 고려하면 될 것이므로 굳이 안정성을 중시하지 않아도 될 것이다. 하지만 구성체를 운영하는 원리는 다르다. 그 구성체에 메여있는 구성원의 수가 하나가 아니기 때문이다. 법의 경우 법이 규율하는 구성체인 사회에 메여있는 구성원의 수가 기본 단위가 몇천만이 되기 때문에, 그들 구성원들의 신뢰의 이익을 보호한다든가 하는 안정성에 신경을 쓰지 않을 수 없다. 프로그래밍 언어 역시 구성체를 운영하는 원리라는 점에서는 마찬가지라고 생각한다. 어떤 한 프로그래밍 언어에 의해 정의되는 프로그램은 수천, 수만, 수억개가 될 수도 있다. 하지만 그 프로그래밍 언어 자체가 안정성이 없이 일관된 결과를 내놓지 못한다든가, 프로그래밍 언어가 발전한다는 명목 하에 갑작스럽게 지원하던 기능을 지원하지 않게 된다든가 하는 식의 일이 발생한다면, 이는 문제가 아닐 수 없다. 프로그래밍 언어가 그 언어를 통해 구현된 프로그램의 신뢰의 이익을 보호하지 못하는 일이 생기는 것이기 때문이다. 따라서 프로그래밍 언어의 발전 및 운영 역시 안정성이라는 가치를 우선에 두고 고려해야 할 것이다.

앞서 값 중심 언어의 고안은 구성체 운영 원리라는 틀에서 보았을 때 ‘혁명’이라는 언급을 한 바가 있다. 물론 패러다임의 교체가 필요하겠지만, 그리고 값 중심 언어는 어느 정도 완충적 역할을 한 바 충분히 인정되지만, 프로그래밍 언어의 세계에서 이러한 혁명적 변화는 다소 위험할 수도 있음을, 이 시각은 전달해주고 있다. 이러한 이유에서 값 중심 언어가 기계 중심 언어의 패러다임 하에서 유효했던, 그리고 쓸모 있는 여러 개념들을 채용하여 지원하고 있다고 생각한다.

5. 나가면서

프로그래밍 언어를 ‘구성체 운영 원리’로 보겠다는 목표를 가지고 프로그래밍 언어의 역사와 법의 계수 과정의 비교검토를 통해 법과 프로그래밍 언어의 유사성을 도출하고, 프로그래밍 언어의 ‘구성체 운영 원리’로서의 의의까지 살펴보았다.

필자는 어떤 주제이든 간에 새로운 시각으로 접근해보는 것이 의의가 있다고 생각한다. 새로운 시각 그 자체가 어떤 의미가 없다 하여도, 어떤 대상에 대해 새로운 시각을 갖는 것은 또 다른 새로운 시각을 불러온다든가 기존의 문제를 해결한다든가 향후 발전을 불러온다든가 하는 식의 의미 있는 일을 충분히 이끌 수 있기 때문이다. 프로그래밍 언어를 ‘구성체 운영 원리’로 보는 것 역시 그런 의미가 있을 것이라고 믿어본다.