프로그래밍 언어 논술 에세이 1

프로그래밍 언어의 과거, 현재 그리고 미래

서울대학교 공과대학 컴퓨터공학부
최 종 식

차 례

1 개요

잘 알려진 프로그래밍 언어의 발전과정과 프로그램이란 무엇인지에 관련된 여러 글들을 읽고, 앞으로 프로그래밍 언어가 나아가야 할 방향성에 대해서 이야기 한다. 또한 프로그래밍 언어의 현재에 대해서 인식하고 논리학에서 시작해서, 계산에 대해 철학의 발전이 뒷받침 되지 않는다면 프로그래밍 언어의 미래를 이야기 할 수 없을 것이라는 사실에 대해서도 이야기 한다. 나아가 궁극적으로 컴퓨터 전공자로서 가져야 할 진보를 위한 객관적이고 비판적인 시각을 키울 수 있도록 한다.

2 서론

2.1 세상에서 가장 잘 알려진 프로그래밍 언어

현재 세상에서 가장 잘 알려진 프로그래밍 언어라고 하면 그 누구라고 해도 C 언어를 뽑는 것을 주저하지 않을 것이다. C 언어를 그리 좋지 않게 생각하는 사람이라고 하더라도 어쩔 수 없이 뽑아야 할 만큼 C 언어는 잘 알려져 있고, 실제로 가장 널리 쓰이고 있는 언어이다. 지금도 C 언어는 발전하고 있는데 2000년 5월에 표준[1]으로 인정받은 “C99”라 불리는 표준이 가장 최근의 것이다.

C 언어는 그 역사에서 살펴볼 수 있듯이 하드웨어의 발전과 그 시스템의 구축을 직접 겪으면서 발전한 언어이고, 따라서 많은 부분에서 기계적인 실체에서 수행되기 위한 여러 가지 개념들을 가지고 있다. 언어의 이식성(portability)을 중요한 목표로 삼지는 않았지만 초기에 수많은 비-호환 컴퓨터에서 수행되던 유닉스 운영체제의 발전과 같이 발전했기 때문에 이러한 이식성은 사람들이 C 언어를 접하고 이용할 수 있는 기회를 획기적으로 늘리는데 기여

1) http://en.wikipedia.org/wiki/C99

했다. 또한 이식성은 언어의 확산과 더불어 표준(standard)에 대한 사람들의 인식을 높여주는 데에도 기여했다. 오늘날C 언어는 유닉스 운영체제뿐만 아니라 광범위한 분야에서 사용되고 있고, 컴퓨터 산업에서 가장 일반적으로 사용하는 언어이다.

2.2 프로그래밍 언어 B

B 언어는 유닉스의 첫걸음을 내딛은 톰슨(Ken Thompson)에 의해 만들어진 시스템 프로그래밍 언어이다. 1960년대 말에 Bell Lab에서는 전망이 좋지 않고 연구비용이 비싸다고 판단해서 Multics 프로젝트를 중단하였다. 이때 톰슨은 프로세스에 대한 개념, 트리형식의 파일시스템 구성과 사용자가 명령을 내리는 명령행 해석기(command line interpreter)에 대한 개념, 각 명령이 새로운 프로세스를 형성해서 실행하도록 하는 개념, 텍스트 파일에 대한 간단한 표현과 장치에 대한 일반적인 접근등 Multics의 혁신적인 면을 대다수 포함하는 대안으로 유닉스를 구상하였다. 하지만 종이 테이프를 읽는 PDP-7 상에서 유닉스의 커널, 에디터, 어셈블러 등의 기초적인 개발을 끝내고 유닉스가 운영체제로서 의미가 있어진 이후에서야 다른 프로그램을 개발할 수 있었다. 그 이전의 모든 것은 어셈블러로 표현되어 있었다. 이러한 하드웨어적인 역경을 견뎌낸 후에 톰슨은 유닉스의 시스템 프로그래밍 언어가 필요하다는 결정을 내리고 그 자신의 언어를 만들었다. 그리고 이것을 B 라고 불렀다. 간단히 말해서 B는 타입이 없는 C라고 보면 되며, 좀 더 자세히는 8KB의 메모리에서 실행되고 톰슨이 BCPL에서 걸러낸 것이다.

2.3 프로그래밍 언어 C

C 언어는 B 언어가 가지고 있는 문제점들을 보완하고 이후 하드웨어 시스템에 대한 대비를 위해서 언어 정의를 확장한 것이다. B

언어는 문자를 다루는 방식, 부동소수점 연산, 포인터 다루기 등에서 문제가 있었고, B 컴파일러의 굉장히 느린 스레드-코드방식도 문제가 되었다. B 언어에서 C 언어로의 이동은 1971년부터 B 언어에 문자열타입을 더하고, 스레드-코드가 아니라 어셈블리 언어와도 그 속도와 크기에서 경쟁할 수 있는 기계명령어를 생성하는 컴파일러를 재제작하는 것과 동시에 이루어졌다. 이후 1973년 초기까지 현대 C 언어의 핵심적인 사항들은 완료되었다. 이 언어와 컴파일러는 PDP-11의 유닉스 운영체제 커널을 그 해 여름동안 재 작성할 수 있을 정도로 강력했다. 이 시기에 컴파일러는 다른 기계를 목표로 삼고 계속 개발되었고, 현대적인 라이브러리를 위한 원형이 완성되었다. 이후 시기에는 unsigned, long, union, enumeration 등의 타입구조를 얻고 유닉스 시스템 유틸리티와 툴들을 재작성하는 등, 언어의 이식성과 타입 안전에 초점을 맞춘 발전이 이루어졌다.

2.4 표준 제정

C 언어의 이식성에 대한 실험이 성공하고 유닉스 운영체제의 대중화와 더불어 인기를 끌게 되자 난립하고 있던 컴파일러를 정리할 필요가 있었다. K&R의 제1판이 나와 있긴 했지만 이것은 명확한 설명을 하고 있지 않았다. 이 과정에서 심지어 pcc조차 K&R의 설명조차 충족하지 못 할 정도로 레퍼런스 컴파일러로의 위상에 부적합해졌다.

표준화 과정은 새로운 이론의 발명 과정도 아니고, D 언어를 만든 작업도 아닌 기존의 C 언어를 자세히 성문화 했던 과정이다. 표준화 과정에서 언어의 핵심은 거의 바뀌지 않았고, 함수의 타입에 인자의 타입을 넣는 변화 이외에는 중요한 변화는 없었다.

2.5 C 언어의 의미와 한계

C 언어는 위 과정에서 보듯 새롭게 만들어진 언어가 아니라 기존의 언어에 하드웨어의 발전과 시대의 흐름에 따라서 덧붙여진 요소들의 결합이라고 볼 수 있다. 그래서 몇 가지 애매한 부분이 있다. 그 예로 포인터와 관련된 타입문제가 있고 모듈화 프로그래밍을 지원해 주지 않는다는 점이 있다. 또한 메모리 관리를 컴파일러가 아니라 프로그래머의 실력에 의존하는 경향이 크다는 것도 문제점이다.

이를 통해 생각해 볼 때 C 언어의 한계는 그 개발과정의 역사성에 기인한다. 첫 번째로 C 언어의 타입에 약한 컴파일러는 애초에 타입이 없는 BCPL과 B 언어에서 발전한 언어라는 역사적인 사실과 무관하지 않다.

두 번째로 C 언어에서 완전히 새로운 언어를 만들기에는 너무 기존의 인프라가 크고 거대하다는 것이다. 언어를 설계하는 사람에게 C 언어는 기존 프로그램 코드와의 호환성을 고려하지 않을 수 없기 때문에 C 언어의 특징인 이식성을 살리기 위해선 어쩔 수 없이 근본적인 구조를 유지해야만 했다.

세 번째로 역사적이라는 말에서도 유추할 수 있듯이 체계적인 이론보다 하드웨어적인 어려움 속에서 발달한 언어인 만큼 C 언어의 발달은 경험에 근거하고 있다. 그 과정을 고려하더라도 직관적이지 않고 작성하기에 애매한 개념들은 그것이 좋은 개념이 아니라는 것을 반증한다. 우리가 짜는 프로그램도 처음에 작성하고 그 뒤로 수정작업을 거치게 되면 굉장히 난해한 코드가 되는 경우를 많이 보았을 것인데 언어의 개념이 이렇게 진화하고 있다면 매우 잘못된 것이 아닐 수 없다. 논리 연산자나 증가/감소 연산자 등의 예에서도 보듯 언어가 처음부터 잘 디자인 된 후 개발된 것은 아니라는 것은 명확하다. 그때그때 필요한 부분을 개발해 나가고 기존의 것들을 조금씩 수정 해 나가면서 발전 해 온 역사는 그 태동이 B 언어의 확장인 것과도 다르지 않은 듯하다.

이후 하드웨어가 눈부신 발전을 이룩하는 동안에도 소프트웨어 산업에서는 어려운 C 언어를 쓰면서/쓸 수밖에 없는 상황이 계속

되었다. 하지만 하드웨어적으로 고급 언어를 구현하고 실행할 수 있는 상황이 되면서 이론적 배경에 입각한 여러 언어들이 개발되기 시작한다.

3 본론

3.1 계산이론

계산이론(theory of computation)은 어떤 문제를 컴퓨터로 풀 수 있는지, 얼마나 효율적으로 풀 수 있는지를 탐구하는 학문이다.[2] 이전의 C 언어가 발전해 온 과정에서 배울 수 있는 것은, 잘 정의되지 않은 언어는 사람의 능력에 따라 좌지우지 되는 경향이 강하고 때때로 문제점이 노출될 때 마다 문제를 그때그때 보완해야 한다는 점이다.

우리는 컴퓨터가 풀 수 있는 문제에 대해서 명확한 정의를 내리고 그에 대비해서 프로그래밍 언어를 설계할 필요가 있다. 컴퓨터는 기계적으로 수행 가능한 어떠한 프로세스도 수행할 수 있다는 것을 알고 있다. 게다가 그런 문제에 대해서 우리가 컴퓨터에게 시킬 수 있는 일들의 한계는 너무나 명확하게도 기계적인 한계보다는 프로그래머로서의 부족함 때문이다.

계산이론은 이런 불문명한 점을 명확하게 정의하고 언어를 명확하게 정의하기 위한 기본적인 이론을 제공한다. 매카시(John McCarthy)의 논문[3]은 이러한 계산에 대한 수학적 이론의 기초를 만드는 것을 목적으로 한다. 첫 번째 파트에서 이 글은 함수를 표현하는 형식에 대해서 정의하고 그것을 실제로 적용하는 예제를 제공하며, 두 번째 파트에서 이러한 표현들의 형식 속성을 언급하고 재귀적으로 정의된 함수들의 동등성을 증명하기 위한 재귀의 유도방법, 정의한 형식이 다른 형식화와 어떤 관계가 있는지를 알

---

2) http://ko.wikipedia.org/wiki/%EA%B3%84%EC%82%B0_%EC%9D%B4%EB%A1%A0
3) *A Basis for a Mathematical Theory of Computation*, John McCarthy, 1963

아본다.

매카시의 글 서문에서도 밝혀져 있듯이 이것은 범용 프로그래밍 언어를 발전시키고, 계산과정의 동등성에 대한 이론을 정의하는 등의 앞으로의 발전을 위해서 밟아나가야 하는 기초적인 과정이다.

3.2 논리학과 프로그램의 정체

계산이론의 발전과 함께 논리학에서도 "프로그램이란 무엇인가"에 대한 증명이 이루어지고 있었다. 세 번째 글[4]에서 그것에 관한 논증을 하고 있는데 저자가 말하는 요지는 그 제목에서 보듯 "증명과 프로그램은 같다"라는 것이고, 본문은 이 명제가 증명되기까지의 역사를 말하고 있다. 프레게(Frege)에서부터 시작하는 현대 논리학은 그림으로 나타내던 많은 증명들을 간단한 기호를 이용해서 나타내기 시작하였다.

겐첸(Gentzen)은 프레게의 판단방법에 "가정"이라는 것을 도입해서 직관적인 추론규칙들을 통해 논리적 증명체계를 구성하는 것이 가능함을 보여주었다. 동시에 전체의 논리적 식들이 언제나 결론이 되는 논리식의 부분으로 단순화 될 수 있다는 하위표현식 이론(subformula property)을 증명하였고 이것을 통해서 상향식(bottom-up) 논리 증명이 가능하게 되었다.

한편 처치(Church)는 기계적으로 계산 가능한 모든 함수를 표현할 수 있는 람다 계산법(lambda calculus)을 제시하였다. 수학에서 함수를 등식으로 표현하는 것과 달리, 함수를 하나의 추상적인 개체로 나타내는 표기법을 통해 모든 계산을 대입으로 환원시킬 수 있다. 람다 계산법에서 재미있는 개념 중 하나로는 함수가 함수를 인자로 받는 다는 발상이다. 이러한 발상은 훗날 여러 언어들에서 유용하게 쓰이게 된다. 이후 처치는 타입 람다 계산법(typed lambda claculus)을 고안 하였는데 람다 항의 타입을 결정하는 타

---

4) *Proofs are Programs: 19th Century Logic and 21st Century Computing*, Philip Wadler, Avaya Labs, 2000

입 도출절차를 포함한 것이다.

위의 두 논문이 동전의 양면과 같이 동일한 내용을 나타낸다는 사실이 1969년에 와서야 호워드(Howard)에 의해서 증명되었다. 단순하게 말해서 증명 절차를 하위 표현식에 의해 단순화 시켜나가는 과정과 람다 추상식(lambda abstraction)에 인자를 대입해서 연속적으로 계산해 나가는 절차와 본질적으로 완전히 동일하다는 것이다.

람다 계산법은 튜링 기계에서 계산 가능한 함수들을 계산하는 체계이기 때문에 람다계산법과 자연 연역(natural deduction) 체계가 동일하다는 것은 기계적으로 계산 가능한 것들의 집합과 논리적으로 계산 가능한 것들의 집합이 동일하다는 것을 나타낸다. 고로 증명은 프로그램이다.

3.3 학문으로서의 소프트웨어

이렇듯 컴퓨터 과학은 수학에 기초를 둔 과학의 한 분야이다. 하지만 단순히 수학이라고 말하기에는 차별화된 영역이 있고 지금도 연구들이 활발하게 진행 중이다. 위에서 이야기 한 이론적인 배경을 바탕으로 실제 시스템의 복잡한 현상들을 분석하고 프로그램(소프트웨어)이 무엇인지 프로그램이 독립된 학문으로서 어떤 위상을 가지는지 그 위치는 어떤지에 대한 연구들이 이루어지고 있다. 계산, 프로그램, 소프트웨어라는 단어는 다르지만 사실은 동일한 분야를 독립된 학문분야로 설 수 있게 하는 개념들이 있다. 하지만 아직 그 중에는 충분히 엄밀한 정의를 바탕으로 정확하게 설명할 수 있는 것도 있지만, 아직도 구름 잡는 논조로 밖에는 설명할 수 없는 것도 있다. 게다가 람다계산법이 그러했듯이 어떤 또 다른 패러다임이 앞으로의 발전을 이끌 가능성이 아직도 충분하다. 학문이 성장했다는 것을 그 개념의 틀이 성숙했다는 것으로 이해한다면, 소프트웨어 학문은 아직도 많은 발전가능성이 있는 학문이다.

4 논의

4.1 현재의 프로그래밍 언어

현재의 대표적인 프로그래밍 언어들은 원시적이고 애매한 부분들이 많고 어떤 면에선 불합리하기까지 하다. 대체 이런 언어를 가지고 어떻게 개발을 해왔을까 하는 의구심이 들 정도이다. 그나마 과거의 언어인 ALGOL 이나 B 언어 등 보다는 진일보한 것이 틀림없지만 아직도 갈 길은 멀다. C 언어는 2.5에서 이야기 했듯이 많은 한계점을 가지고 있고 이제 와서는 손을 쓰기가 힘들 정도로 고착화 되었다. 물론 그 와중에서도 컴퓨터 과학은 눈부시게 발전했지만 이것은 마치 모래로 쌓은 성과 같다. 블루 스크린으로 대표되는 소프트웨어의 치명적인 버그들로 인하여 우리가 쓰고 있는 컴퓨터가 다운되는 일이 아직도 많고 우리는 우리가 작성한 프로그램이 어떻게 동작할지 정확하게 예측하지 못한다.

이것은 멀리 볼 것도 없이 전기-컴퓨터 “공학”이 매우 정확한 이론적 기반 위에서 표준을 정하고 하드웨어를 발전시켜 온 것과도 대비된다. 덕분에 과거에 이론이 현실에서 좌절해서 조악한 언어를 만들었던 것과는 달리 이제는 제대로 된 이론적 기반 위에서 언어를 설계하고 컴퓨터를 발달시킬 수 있는 하드웨어적 기반이 마련된 것이다.

4.2 프로그래밍 언어 담론

지금까지의 프로그래밍은 언어에 사람이 맞춰서 온 것이라고 생각한다. 프로그래머가 세밀한 메모리 관리를 해 주는 것은 물론이고 하나하나의 타입도 섬세하게 지정해 줘야 했다. 그러고도 혹시 모를 버그에 두려워해야한다. 현대에 와서야 비로소 이전에 사람들이 연구하고 꿈꾸던 프로그래밍 언어에 대한 연구가 빛을 발하고

있고 언어다운 언어에 대한 담론들이 이루어지고 있다.

프로그램을 만들 때 그 도구가 제대로 되어 있지 않다면 제대로 된 프로그램이 작성될 리가 없다. 프로그램이 발매 된 후에 베타버전, 안정화버전, 버그패치 등의 여러 가지 과정을 거치는 이유가 다른 곳에 있는 것이 아니다. 또한 버그의 문제도 도구의 문제에서 귀결된다고 본다. 버그에 대한 명확한 정의와 범위, 그리고 타입 문제 등이 제대로 고려되지 않은 도구를 사용하는데서 오는 것이다.

모든 소프트웨어 시스템의 경쟁력에는 프로그래밍 언어의 발달을 기초로 하는 부분이 들어있다. 언어 없이 작동하는 소프트웨어 시스템은 없다. 0101001과 같은 전기적 신호도 언어의 일종이고 어떤 형태이든 언어를 기반으로 한다. 때문에 그 언어가 제대로 되어있지 않다면 시스템을 구축해도 어떤 문제점이 나타날지 모르는 두려움을 가져야 하는 것이다.

궁극적으로 프로그래밍 언어는 프로그램(시스템)을 완성을 한 후에 내가 의도한 동작을 한다는 확신을 가질 수 있는 도구가 되어야 할 것이다. 소프트웨어 이외의 분야에서 설계 이후에 제대로 작동할지를 미리 검증하는 기술이 없는 분야는 없다. 비록 문법검증, 타입검증 등의 기술이 개발되었지만 아직 미흡하다. 이런 검증 기술만으로는 부족하다. 이 문제에 대한 답은 언어의 설계부터 해주어야 한다.

5 결론

5.1 이론에 기반을 둔 프로그래밍 언어 개발

지금의 언어가 가지고 있는 문제점들을 볼 때 미래 프로그래밍 언어가 나아가야 할 방향은 너무나 명확하게도 이론적 기반을 둔 언어이다. 엄밀한 정의와 명확함을 기초로 하는 수학, 논리학에 기반을 둔 컴퓨터과학이 발전하는데 그런 엄밀한 이론을 빼 놓는다

는 것은 말도 안 된다. 이는 역사란 현재의 틀에서 과거를 반성하고 타산지석으로 삼아 미래를 열어가는 작업임을 생각해 볼 때에도 마찬가지다. 대표적으로 람다 계산법이라는 이론적 배경은 프로그래밍 언어 개발의 패러다임을 바꾸었다.

하지만 어떤 이론적 개념이 프로그래밍 언어 설계에 도움이 되는 것인지 아닌지를 판단하는 이론도 필요하다. 모든 이론적 개념이 언어 설계에 도움이 되는 것이라고 보기에는 힘들기 때문이다.

프로그래밍 언어가 가져야 하는 어떤 개념, 속성과 추구해야 하는 지점에 대한 명확한 합의가 먼저 필요할 것이다. 지금도 C 언어와의 전투는 계속 되고 있고, 그들의 의견도 일견 타당한 점이 있는 것은 사실이다. 아마도 올바른 개념적 프로그래밍 언어를 위해서 이 분야뿐만 아니라 하드웨어 분야도 바뀌어야 할지도 모르는 일이다. 합의를 도출 한 뒤 그것을 위한 이론체계를 찾아보거나 연구해 보는 것도 충분히 의미가 있는 일이라고 본다.

우리는 아직도 프로그래밍에 대한 본질을 완전히 이해하고 있지 못하다. 이론 체계는 이러한 잘 모르는 본질에 대해 이해하는 의사결정 과정을 보다 체계적으로 만들어 줄 것이다. 또한 기계적 계산의 또 다른 의미를 연구해보면 완전히 새로운 패러다임이 또다시 제시 될지도 모르는 일이다. 프로그래밍 언어는 아직도 미개하다. 때문에 아직도 우리의 할 일은 많다.

5.2 컴퓨터를 전공하는 학생

각 각의 글들은 다른 이야기를 하고 있는 글이지만 하나로 연결되는 글이다. 잘 알려진 언어의 역사를 살펴볼 수 있고, 프로그램을 기술하는 프로그래밍 언어가 지향해야 할 점을 살펴보기 위해서 그 형식을 정의하고, 튜링 머신으로 대표되는 기계적으로 계산 가능한 프로그램이 사실은 논리학의 연역법과 동일하다는 사실을 이야기 한다. 이를 읽고 난 후에 전에 비해서 현재까지 내가 써 오던 언어들에 대해서 좀 더 객관적인 시각을 가지게 되었다. 사실

이전까지는 버그가 나고 오동작 하는 것을 프로그래머의 실력 탓이라고 생각했는데, 그게 문제가 아니라 처음부터 도구가 올바르다면 그런 일이 일어나지 않을 것이라는 생각을 하게 되었다.

존재하는 것을 의심 없이 받아들이는 것이 아니라 다른 문제점이 없는지, 어떤 점은 불합리한지, 궁극적인 목표는 어디로 가야하는지에 대한 생각도 할 수 있었다. 내가 이때까지 써오던 C 언어를 다시 보는 것은 물론이고 nML은 올바른 언어일까? 차세대 프로그래밍 언어는 어떤 언어일까? 하는 등의 여러 가지 생각을 할 수 있게 만들어 준 글들이다.

나는 언어의 사고방식에 나를 맞추는 것이 아니라 언어를 인간의 사고에 맞추어야 한다는 생각에 동의한다. 다만 과학의 측면뿐만 아니라 공학의 측면에서도 언어 설계를 고려해야 한다는 것이, 이 과정을 역사적 사실과는 또 다른 의미로 발달에 장애물이 될 수도 있을 것이라는 생각을 한다.

하지만 그 어떤 소프트웨어 시스템에도 언어적 이론이 포함되어 있고 그러한 시스템을 내가 개발한다면 무엇보다 안전하고, 직관적이며 개념적으로 올바른 소프트웨어를 만들고 싶다. 이를 위해서 아직도 내가 할 일은 많다. 그런 언어의 개발을 소프트웨어 학문의 초기시대를 살아가는 바로 내가 할 수도 있는 것이다.

6 참고문헌

[1] *The Development of the C Language*, Dennis M. Ritchie, HOPL-II, 1993

[2] *A Basis for a Mathematical Theory of Computation*, John McCarthy, 1963 (1996)

[3] *Proofs are Programs:19th Century Logic and 21st Century Computing*, Philip Wadler, Avaya Labs, 2000

[4] *계산이란 무엇인가에 대한 아이디어들*, 이광근, 1999