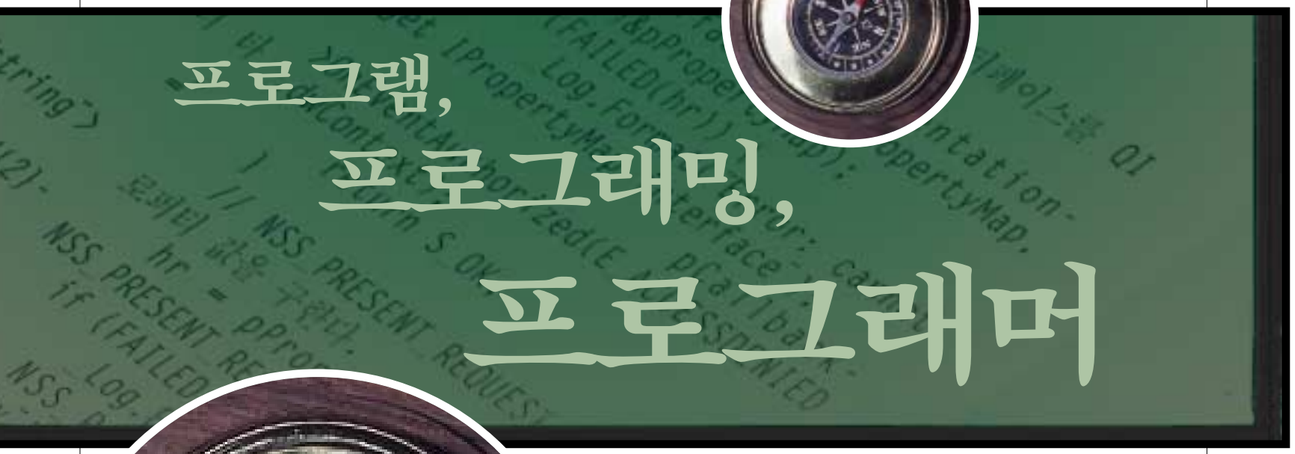
프로그램, 프로그래밍, 프로그래머

김창준 중앙대 컴퓨터공학과 juneaftn@hanmail.net

이 글은 프로그램과 프로그래밍, 프로그래머에 대한 매우 '일반적'인 글이다. 즉, 특정 언어의 최적화 기술이라든가, 특정 도구의 활용법 같은 특수하고 한정된 대상에 국한된 것이라기보다는 일반적인 프로그래밍과 그 수련에 대한 것이고, 이것은 좀더 보편적으로 모든 지식에 일부 적용될 수 있다. 따라서, 이 글은 '일반적 지식론 적용의 한 가지'라고 불러도 무방할 것이다. 즉, 여기에 나오는 '프로그래머론'은 철저하게 '일반 지식론'의 성립 하에서 자연스레 도출된 것이며, 어느 한 부분도 '일반 지식론'을 조금도 부정하지 않는다.

우리는 프로그래밍을 통해 프로그램을 만드는 프로그래머들이다. 하지만 이런 기본적인 것들을 무시한 채 지내는 것에 익숙해진지 이미 오래다. '프로그래머' 자의식의 종말과 왜곡을 목전에 두고, 다시금 출발점으로 돌아가 우리의 존재론적 가치와 지향점 등에 대한 화두를 회복해야 할 것이다. 인간이란 유일하게, 자기 자신과 자신이 하는 일을 관찰하고 반성할 수 있는 동물이기 때문이다.



또 이 글은 어떤 것이 '좋은' 프로그램인지, 누가 '훌륭한' 프로그래머인지에 대한 근본적인 판단근거와 기준을 제공할 것이며, 이러한 기본적인 것들에 대한 여러분 스스로의 관심을 촉구하는 데 목적이 있다. 따라서 글에서는 주로 '뼈대'에 해당하는 것을 다룰 것이며, 그 설명을 위한 방편으로서의 '살'이 들어올 수는 있으나 결코 이 글 자체가 그런 '살'의 구체적 설명을 위한 것이 아님을 미리 밝힌다. 다시 말해 화면에 라디오 버튼을 출력하는 방법이라든가, 패킷을 보내는 방법, 또는 멀티쓰레딩을 할 때 커널에서 어떤 변화가 일어나는지 등에 관한 내용은 기대할 수 없을 것이다.

서두가 조금 위압적이고 단언적인 선언문의 꼴이 돼버렸는데, 독자들에게 고자세를 취하거나 겁을 주려는 의도는 전혀 들어있지 않다. '뿌리'가 확실해야만 그 '가지'가 바로 자랄 수 있다는 생각에 다소 엄격한 언어를 사용한 것 뿐이다. 사실 이 분야에서는 '가지'를 다루는 글이 잘 팔린다. 물론 실상 지면에 쏟아져 나오는 대다수가 가지나 이파리류의 글이다(이런 것도 분명 중요한 작업이긴 하지만). 이로 인해 이리저리 유행에 휩쓸리는 것도 많이 보게 된다.

조금 동떨어진 얘기일지 모르겠으나, 유행(fashion)이란 말이 나온 김에 패션 이야기를 잠시 해야겠다. 필자가 가장 존경하는 패션 디자이너로 가브리엘 본헤어 샤넬(Gabrielle Bonheur Chanel)이 있다. 그는 21세기를 사는 우리가 입는 옷의 기본 구조를 결정한 사람이자 패션의 교과서다. 한마디로 삶의 양식과 문명을 변화시킨 사람이다. 세상에 끼친 그의 영향이 이렇게 클 수 있었던 것은 오히려 일시적 유행에서 동떨어져 있었기 때문이 아닐까 한다. 강호에서 유행에 맞춰 나가는 디자이너는 부지기수다. 하지만 자신의 모드를 고수하며 유행에 물들지 않고 나름의 지속적 변화를 꾀하는 디자이너는 소수다. 그들이 세상을 바꾼다. 그들은 근본적으로 '옷'과 '패션'에 대한 화두를 늘 놓치지 않는 사람들이다. 아직까지도 서점에서 자신의 위용을 뽐내고 있는, 칠팔 십 년대 이전에 초판으로 출간된 컴퓨터 서적들을 알고 있는가. 불과 일이 년만 지나도 고서 신세가 되고, 하루에도 수 십 권의 책들이 명멸하는 컴퓨터 서적의 전국시대에 무엇이 이토록 오랜 가치를 지니도록 했을까. 필자는 이것이 근본과 뿌리에 대한 지속적인 반성에서 비롯됐다고 본다.

어원학적 고찰

세계에서 가장 많이 팔리는 예술 서적은 곰브리치(E. H. Gombrich)의 「예술 이야기(The Story of Art)」다. 예술 전반에 걸쳐 역사적인 흐름을 따라가며 그림과 함께 독자들이 예술을 직접 느끼고 체험할 수 있도록 도와주는, 그야말로 예술 개론서로는 '탁월한' 책이다. 그런데 이 책의 서두부터가 심상치 않다.

"사실 예술이라고 하는 것은 없다. 오로지 예술가들이 있을 뿐이다."

웃기지 않은가. 예술 이야기라는 책의 첫 문장이 '예술은 없다'는 것이니 마치 종교 경전에서 첫 문장이 '신은 죽었다'로 시작하는 것과 별반 차이가 없어 보인다.

곰브리치는 여기서 대명사로서의 예술, 즉 첫 글자가 대문자인 예술(Art)을 지칭한 것이다. 이것은 어떤 고정적이고 단일한, 천상 어딘가에 그 완벽한 원형이 고이 간직되어 있을 존재로서의 예술은 없음을 의미한다. 어떤 그림을 보고 좋아하는 아이에게 "네가 지금 좋아하는 그건 예술이 아냐!"라고 핀잔을 주는 것만큼 잔인한 행위도 없다. 물론 더 잔인한 것은 예술가가 고통 속에 출산한 예술품을 두고 "이건 예술이 아냐!"라고 외치는 대중일지도 모르겠다. 결국, 곰브리치의 말은 한마디로 예술이 그것을 행하는 사람 이전에 어떤 고정된 실체로 존재하는 것은 아니라는 것이다. 예술을 먼저 실체화하여 설정하는 것은, 손가락 개수가 다섯 개인 이유가 장갑에 뚫린 구멍이 다섯 개이기 때문이라는 궤변일 뿐이다.

그렇다면 이 분야에서는 어떨까. 누군가가 몇 시간을 공들여 만들어 놓은 것을 보고 "이건 프로그램이 아냐!"라고 말할 수 있을까. 이 질문 이전에 '과정'으로서의 '프로그래밍'이란 무엇이고, 거기서 나오는 '결과'로서의 '프로그램'은 무엇이며, 이 작업을 행하는 '주체'로서 '프로그래머'란 누구인가.

이 세 단어의 가장 원형이라고 생각되는 '프로그램(program, programme)'이라는 단어가 역사적 문헌에서 최초로 등장한 것은 17세기(1633년)까지 거슬러 올라간다. 인류가 이 단어를 사용한 역사는 불과 400년도 되지 않는다. 초기에는 '미리 쓴다', 즉 뭔가를 미리 써서 알리는 '사전 고지'를 의미했다. 이것은 미리 계획을 세워서 알리는 개념으로 발전했고 나중에는 그 계획 자체를 뜻하게 되었고, 1920년대 들어 기술의 발전으로 라디오가 탄생하면서 라디오 진행표를 일컫는데 사용되기에 이르렀다. 우리가 말하는 컴퓨터 프로그램으로서의 '프로그램'이 최초로 등장한 것은 1946년 네이처(Nature)의 에니악(ENIAC) 컴퓨

터 관련 기사였다. 한편 프로그래머라는 단어가 문헌에 등장한 것은 1890년 경마 클럽의 '프로그래머'가 시초였고, 컴퓨터 프로그래머는 그보다 한참 후인 1948년도였으며, 이것이 사람들 앞에서 밝힐 수 있을 만큼 '떳떳한' 직업이 되기까지는 수십 년이 더 흘렀다. 튜링상을 받은 다익스트라는 1957년 결혼 당시 자신의 직업란에 프로그래머라고 적었다가 "이런 직업은 존재하지 않는다"는 반론에 이론 물리학자라고 고쳐 썼다고 한다.

프로그래밍이라는 작업은 비교적 이른 시기에 찾아볼 수 있는데, 1949년 Mathematical Tables & Other Aids to Computation이라는 저널에 재미있는 용례가 있다.

"ENIAC 사용에서 발생하는 가장 시간 소모적인 요소는 프로그래밍(특정 문제를 위해 기계를 준비하는 것)이다."

아마 소프트웨어 위기(Software Crisis)는 컴퓨터가 출현한 초기 시대부터 예견됐는지도 모르겠다. 반세기가 지난 지금도 우린 그 시간 소모적인 '프로그래밍'의 문제를 해결하지 못하고 있다.

어원연구에서부터 어휘 사용의 역사까지 살펴봤으니 현재적 의미에서 합당한, 새로운 정의를 이끌어 내야 할 것 같다. 이것이 온고지신 아니겠는가. 하지만 필자는 앞서 곰브리치가 지적한 것처럼 '완벽한 프로그래밍의 정의'를 닫힌 괄호 속에 채워 넣는 식의 어리석음을 범하고 싶지는 않다.

프로그래머란 프로그램을 만드는 사람을 말한다. 프로그래밍이란 프로그램을 만드는 작업을 말한다. 이런 식으로 두 개의 단어는 제대로 정의가 된 것처럼 보인다[주: 조금 약삭빠른 것 같기도 하다]. 프로그램이란 "문제 해결의 단계를 코드화해 표현한 것"이라고 하면 어떨까. 현존하는 대다수의 컴퓨터가 폰 노이만식이니 단계 이야기를 꺼내도 될 것이고, 프로그램이라는 말 자체는 본질적으로 인간의 관점에서 본 것이니(논쟁의 여지가 있긴 하다) "문제 해결" 따위의 말을 써도 될 듯 하다. 또 정해진 표현양식과 유한집합의 명령어가 있으므로(이게 아니면 알고리즘이라고 해야할 것이다) '코드화'라는 표현을 사용하는 것은 적절하다. 어쨌든 컴퓨터 프로그램도 '미리 써둔 계획'의 일종이니 몇 백년도 넘은 단어가 처음 의미를 크게 벗어나진 않은 것 같다.

그렇다면 이런 말장난 같은 것들로부터 도대체 어떤 쓸모를 얻을 수 있을까. 위의 정의를 고려해 볼 때, 일반적인 의미에서 프로그래머는 '문제 해결자'라고 말할 수 있다. 흔히들 프로그래머간에 자조적으로 자기를 폄하하는, 동시에 책임회피에 아주 유용하게 사용하는 문장이 있다. "전 단지 프로그래머여요(I'm just a programmer)"라는 문구인데, 이 글을 읽는 순간 당장 생각을 바꿔야 한다. "(놀라지 마세요) 저는 프로그래머랍니다"로 말이다. 자신이 설령, 시스템 분석가나 아키텍트가 던져주는 문서를 보고 단순 노동으로 프로그래밍하는 사람일지라도 자존심을 가져라. 이미 그들은 '일반적 문제 해결자(general problem solver)'의 배경과 가능성을 갖고 있는 것이나 마찬가지이기 때문이다.

같은 뜻, 다른 표현

문제를 해결한다는 것은 무엇인가. 수학 문제를 푸는 것은 무엇인가로 생각의 범위를 좁혀 보자. 모든 수학 문제는 기본적으로 경험적이 아니라 분석적이다. 그것은 세종대왕이 한글을 창제한 해와 같이 어떤 직간접 경험을 통해 알 수 있는 것이 아니고, 그 문제 자체와 몇 가지의 수학적 공리를 분석하면 선험적으로 알 수 있다는 말이다. 한마디로 문제를 완벽하게 이해하고, 기본이 되는 공리를 알고 있다면 답을 알 수밖에 없다는 것과 같다.

예컨대, $f(x)=x^2-3x+2$일 때, $f(x)=0$이 되는 실수 x를 묻는 경우, 문제의 $f(x)=x^2-3x+2$를 $f(x)=(x-1)(x-2)$로 변형하면 답은 거의 자명해진다. 결국 모든 수학문제의 답은 문제의 조건을 변형한 것으로 귀착된다. 이 변형은 어디를 지향점으로 할까. 당연히 인간에게 명백하고 직관적인 수준(혹은 수학적 공리의 수준)으로의 변형을 목표로 한다.

1978년에 노벨 경제학상을 수상하고 심리학과 컴퓨터의 인공지능 분야에 지대한 공헌을 한 허버트 사이먼(Herbert A. Simon)은 그의 대표적 저서 '인공 과학(The Sciences of the Artificial)'에서 "문제를 푼다는 것은, 그것을 해답이 드러나도록 표현하는 것"이라고 언급했다. 그가 예로 드는 '표현 양식(representation)의 변형에서 답이 자명해지는' 대표적인 경우를 하나 보도록 하자.

카드 한 벌에서 하트 문양의 1부터 9까지의 카드를 준비한다. 이 카드를 두 사람 사이에 숫자가 보이도록 일렬로 벌여놓고 두 사람이 번갈아 가며 한 장씩 카드를 고른다. 이렇게 해서 자신이 모은 카드의 숫자 합계가 먼저 15가 되는 사람이 승자가 된다. 이 게임에서 이기거나 최소한 비기기 위해서는 어떤 전략을 써야할까. 그다지 쉬운 문제는 아닌 것 같다. 그런데 좀 전에 f(x)를 인수분해해 변형했듯이 현재 문제의 표현 방식을 다르게 생각해 보자.

〈그림 1〉은 어느 쪽으로 더해도 합계가 15로 일정한, 소위 마방진의 하나다. 문제

〈그림 1〉 마방진

4	9	2
3	5	7
8	1	6

를 이렇게 표현한 경우, 카드를 뽑는 게임은 자신의 말을 일렬로 먼저 늘어놓는 사람이 이기는 틱택토(Tic-Tac-Toe) 게임[주: 서양에서 매우 흔한 게임으로, 가로 세로 세 칸씩 총 아홉 칸의 상자에 두 사람이 한 번씩 번갈아 가며 자신의 표시(○ 또는 ×)를 그려 넣는다. 한번 채워진 곳에는 다시 그려 넣을 수 없고, 먼저 자신의 표시를 일렬로 채운 사람이 이기게 된다. 종종 'Hello! world' 프로그램과 함께 표준 테스트 중 하나로 인정받는다]과 완전히 동일해진다.

〈그림 2〉 틱택토 게임

×	9	2
○	○	×
8	1	6

따라서, 같은 문제라도 그 표현 방식에 따라 쉽고 간단한 문제가 되기도 하고 그 반대가 되기도 한다. 우리는 문제를 적절한 표현 방식으로 바꿈으로써 문제를 즉각적으로 풀어낼 수도, 혹은 문제 풀이에 큰 도움을 받을 수도 있는 것이다. 예컨대 로마 숫자와 아라비아 숫자를 비교해 보면 전자의 경우 덧셈에서는 후자보다 더 직관적일 수 있다(덧셈의 계산이 바로 물리적으로 문자를 더하는 데에서 행해지기 때문이다). 하지만 곱셈이나 나눗셈의 경우 로마 숫자는 비효율적이고 어렵다(로마인들은 이것을 해결하기 위해 산술 연산의 경우, 문자간의 계산을 돌맹이간의 계산, 즉 주판이라는 새로운 표현 방식으로 바꿨다).

수학적으로는 문제를 어떻게 변형하든지 그 의미론은 동일하다. 또 그래야만 한다. 다만 표현 방식이 바뀔 뿐이다. 이러한 것들이 프로그래밍에서는 어떤 의미를 가질까.

컴퓨터의 이론적 배경

이야기를 진행하기에 앞서 독자의 이해를 돕기 위해 몇 가지 기본적 사항을 짚고 넘어가도록 하자. 기본적으로 문제를 푼다, 계산한다(compute)고 하는 것은 수학적인 의미에서의 함수를 말한다. 어떤 값 x를 f(x)라는 함수에 입력했을 때 그 결과값이 무엇인지를 알아내는 것이다. 여기서 x는 단순히 숫자만이 아니라 훨씬 다양한 대상(글자, 문장, 그림 등)이 될 수 있다. 물론 컴퓨터는 이를 단지 기호(symbol)의 일종으로만 여길 뿐이다.

우리가 '컴퓨터'라고 하는 것의 이론적 모델은 튜링 기계(Turing Machine)다. 앨런 튜링(Alan Turing)이 1936년 발표한 이 기계는 실재하지 않는 이론상의 가상적 존재로, 현존하는 모든 컴퓨터와 앞으로 나타날 모든 컴퓨터(현재 컴퓨터의 패러다임을 유지하는 한)의 이론적 한계를 긋는 '가장 강력한' 컴퓨터다. 여기서 가장 강력하다는 것은 계산 가능성(computability)을 지칭하는 것으로, '얼마나 빨리' 또는 '얼마나 효율적으로'와 전혀 상관없이 유한한 시간 안에 이 문제를 풀 수 있느냐는 가불가의 판단이다. 즉, 튜링 기계가 풀어낼 수 없는 문제는 어떤 컴퓨터도 풀 수 없고, 역으로 슈퍼컴퓨터가 풀 수 있는 문제라면 튜링 기계도 풀어낼 수 있다.

튜링 기계는 정해진 길이의 기호(예컨대, 0이나 1)를 적어 넣을 수 있는 칸(cell)이 있는 무한한 길이의 테이프와, 그 테이프에 기호를 쓰고 읽을 수 있는 헤더, 처리 장치로 이뤄진다. 처리 장치에는 유한한 개수의 상태 중 하나를 저장할 수 있는데, 이 상태와 테이프 위에서 현재 헤더가 위치한 곳의 기호에 따라 다음의 행동이 결정된다. 가능한 행동은 헤더를 좌우로 한 칸 움직이는 것, 현재 칸을 읽고 다시 기호를 써넣는 것, 실행을 종료하는 것이 있다. 이렇게 한 단계가 끝나면 현 상태를 새로운 상태로 바꾸고, 다시 다음 단계를 실행한다. 아주 단순한 기계 같지만 무한한 길이의 테이프와 가장 기본적인 몇 가지 동작으로 알고리즘이 존재하는 문제는 모두 풀 수 있다.

오늘날 컴퓨터의 실질적 모델은 폰 노이만 구조(Von Neuman Architecture)를 따른다. 폰 노이만 구조는 기억 장치(메모리)에 자료(데이터)와 명령어를 함께 저장하고, 중앙 처리 장치가 그 저장된 프로그램의 명령어를 하나씩 읽어와서 자료에 대해 연산하는 구조를 말한다.

컴퓨터는 근본적으로 오직 한 가지 언어만을 이해한다(하지만 어떤 한 종류의 기계어가 모든 컴퓨터에 두루 쓰이는 공통어는 아니다). 우리는 이것을 기계어라고 부른다. 이 언어는 2진수로 표현되고, 명령어(연산자)와 목적어(피연산자)가 짝을 이룬다. 예를 들어, 기억 장소의 세 번째 칸에 있는 값을 하나 증가시키라는 식이다.

인간은 이런 2진수로 표현된 명령어와 자료를 이해하기 힘들고 이 언어로 표현하기도 어렵다. 따라서 좀 더 인간의 자연 언어에 가까운 언어를 개발해 사용하게 된다. 이를 고수준 언어라고 부른다. 하지만 컴퓨터는 이것을 이해하지 못하기 때문에 번역의 과정이 필요하다. 고수준 언어로 된 프로그램 전체를 한번에 기계어로 번역해(이 번역 행위 역시 기계어로 작성된 프로그램

이 한다) 이를 차후에 다시 사용할 수 있게 저장하는 것을 컴파일(compile)이라고 하고, 한 문장씩 번역해 순서대로 실행하는 것을 인터프리트(interpret)라고 한다. 고수준 언어에서 명령어 하나는 저수준 언어(기계어)의 명령어 수 십 개에서 수 천 개 이상으로 구성된 것이다.

컴퓨터 세계의 어족

이 세상에 존재하는 프로그래밍 언어는 줄잡아서 2000개가 넘는다. 하지만 기본적으로 범용언어인 경우, 대부분 그 표현능력이 동등하다{주: 쳐치-튜링 논제(Church-Turing Thesis)라고 한다. 순서대로 진행하는 과정(알고리즘)으로 정의되는 함수에 튜링 기계로도 계산할 수 없는 것은 있을 수 없다는 논제다. 따라서 튜링 기계를 흉내낼 수 있는 언어라면(대부분의 범용언어는 튜링 기계의 단순한 동작을 흉내낼 수 있다) 어떤 알고리즘이든지 표현할 수 있다. 물론 기억장치의 한계를 고려하지 않았을 때의 얘기다}. 갑이라는 언어가 계산할 수 있는 것(혹은 풀 수 있는 문제)이라면 을이라는 언어도 할 수 있다는 말이다. 차이점이 있다면 어떤 표현 방식을 사용하느냐이며, 이는 곧 어떤 문제의 답을 구하는 방법(알고리즘)을 표현할 때 효율적이냐 비효율적이냐의 차이로 나타난다.

눈이 많이 내리는 지방에는 눈과 관련된 표현이 발달하고 추상화(경제적 의미에서)가 많이 진행되는 반면, 사막 지방에는 낙타와 관련된 단어가 많다. 자연언어의 이러한 차이는 인공언어인 컴퓨터 프로그래밍 언어간에서도 관찰할 수 있다.

몇 가지 특징에 의해 프로그래밍 언어를 묶을 수 있는데, 소수의 한정된 디자이너가 존재하는 이유로 각각이 독특한 '언어관'을 내포하는 경우도 흔하다. 이런 이유로 프로그래밍 언어의 어족 개념을 '프로그래밍 패러다임{주: 이에 관해서는 1978년에 튜링상을 수상한 로버트 플로이드 교수의 「The Paradigms of Programming」을 참고하라}'이라고 부르기도 한다. 각각은 해당 언어로 생각하는 사람들을 특정 표현 방식으로 생각하도록 유도한다. 언어와 사고에 대해서는 조금 후에 다시 논의하도록 하자.

이 프로그래밍 패러다임은 크게 네 가지 범주를 벗어나지 않는다. 명령형(imperative), 함수형(functional), 논리형(logic), 객체지향형(object-oriented) 언어가 그것이다.

각각 나름의 특징과 장단점이 있기 때문에 다른 것과는 확연히 구별된다. 하지만 이들 언어가 실행되기 위해서는 컴퓨터가 이해할 수 있는 유일한 언어인 기계어로 번역이 돼야 하며, 튜링 기계에서도 언급했듯이 각 언어의 표현능력(expressivity)은 기본적으로 동등하다.

결국 기계어로 번역될 것임에도 불구하고 기계어 대신 고수준 언어(3세대 언어라고도 한다)를 사용하는 것은 이것이 인간에게 더욱 가깝고, 추상화가 잘 돼 있어 자잘한 것들에 신경 쓸 필요가 없기 때문이다. "밥을 먹어" 대신에 "수저에 밥알 220알을 퍼서 상방 30도 각도로 올린 다음 입안에 넣고 씹는 운동을 하는 것을 밥그릇이 빌 때까지 반복하라"는 명령을 해야 한다는 것은 악몽이다. 훌륭한 표기법은 우리 뇌의 모든 불필요한 작업을 덜어주고 우리가 더 고차원적인 문제에 집중할 수 있도록 해주며 결과적으로는 인류의 정신력을 증가시켜주는 것이라고 한 철학자 화이트헤드(A. N. Whitehead)의 명언은 고스란히 프로그래밍 언어의 세계에도 적용된다.

이미 말했듯이 각 언어의 표현능력이 동등함에도 불구하고 여러 언어가 존재하는 것은 각 문제와 상황에 더욱 적당한 언어가 있기 때문이다. 물론 여러 패러다임을 모두 갖춘 언어 하나를 사용하는 것이 최선일 것 같지만 컴퓨터는 그렇게 유연한 기계가 되지 못한다(여러 패러다임을 갖춘 언어를 멀티 패러다임 언어라고 하며, 극히 소수가 존재한다). 따라서 여러 가지 도구를 다룰 줄 알면서 상황에 맞게 적절한 것을 골라 쓰는 장인의 지혜가 필요하다.

우리가 이렇게까지 많은 프로그래밍 언어를 만든 것은 궁극적으로는 자연언어를 모방해 보려는 다각적 시도였는지도 모른다. 프로그래밍 언어에서 자연언어를 제5세대 언어라고 부르며, 오르지 못할 바벨탑에 도전하는 어리석음으로 매달리고 있는 것도 결국은 컴퓨터에서 인간성을 찾기 위한 노력이지 않을까.

프로그래밍 언어와 사고

"우리가 만약 다른 언어로 말한다면, 우리는 다소 다른 세상을 인지할 것이다.- 비트겐슈타인"

언어가 우리의 사고를 형성하는 것일까, 아니면 우리의 사고가 언어를 형성해 나가는 것일까. 닭과 달걀의 논쟁처럼 끝이 보이지 않는 우문같이 들리지만, 굳이 답을 찾는다면 어떻게 말해야 할까. 우리의 언어와 사고는 상호 작용을 하는 양방향의 관계에 있다. 마주보는 거울을 생각하면 된다. 한쪽 거울에 비친 상이 다시 반대쪽 거울에 비치고, 그 (거듭) 비친 상은 또 다시 원래의 거울에 비치는 식으로 무한히 연속된다.

언어는 틀이고 툴(tool)이다. 인식의 도구이자 인간을 종속시킨다(우리는 사용하면서 동시에 지배받는다). 우리는 언어로 세계를 형성하고 이해한다. 다른 언어를 사용하는 이상, 같은 상황을 보면서도 다르게 인식할 수밖에 없다{주: 언어가 우리의 사고 방식을 결정한다고 하는 것을 언어 결정론이라고 한다. 사피어 훠프(Sapir-Whorf) 가설이 이 이론을 주장했고, 20세기 언어학에 많은 영향을 끼쳤지만 현재는 학계에서 그 자체로 받아들여지지 못하고 있다. 사피어와 훠프는 어떤 개념이 특정 언어에만 존재하는 것에서 출발해 해당 언어 사용자의 사고의 고유함을 말했지만, 그러한 경우에도 충분히 성공적인 번역이 가능하다는 점이 이 가설에 대한 가장 큰 반격이다. 필자 역시 이 가설에서 한 걸음 물러선 입장이며, 우리의 삶과 그 양식이라는 좀더 큰 그림을 봐야 한다고 생각한다}. 마샬 맥루한의 말에 빗대어 "언어는 메시지다"라고 말하고 싶다. 즉 어떤 언어를 사용하느냐에 따라 그 메시지가 달라질 수 있다는 말이다.

하나의 인간 개체가 태어나면서부터 겪게 되는 언어 발달 과정을 유심히 관찰하면 우리 인류가 겪어온 언어 발달의 역사를 유추해 볼 수 있다(부분적 마이크로 모델에서 전체의 매크로 모델이 유추되는 것이다). 처음 글자가 없을 때의 사고 구조(구술 문화)와 글자가 생기고 나서의 사고 구조(문자 문화)의 차이는, 글자는 모르지만 말은 하는 아이와 나중에 글과 말을 모두 터득한 아이의 비교를 통해 대강 짐작할 수 있다. 한 가지만 예를 들면, 구술 문화의 사람들은 말의 호흡이 짧다. 이것은 인간의 한정된 기억 능력에 기인한다. 반면, 글자가 사용되면서부터 자신의 뇌 외부에 말을 기록하는 것이 가능해졌고, 이는 말의 호흡을 길게 하는 효과를 가져왔다(이것은 언어, 나아가 그 사람의 사고 활동이 좀더 복잡해지고 추상화되도록 한다). 마찬가지로 글을 모르는 아이와 글을 사용하는 아이의 언어 차이도 이와 유사하다. 결국 어떤 언어를 사용한다는 것은 그 사람에게 사고의 어떤 틀을 제공한다는 말과 같다. 물론 그런 언어도 해당 언어 사용자들의 사고 구조에 가장 적합한 형태로 진화하고 있다. 언어와 사람의 사고는 이처럼 불가분 관계에 있다.

우리가 한 가지 언어를 배운다는 것은 그 언어를 사용하는 사람들의 사고 구조를 몸에 익힌다는 말과 크게 다르지 않다. 영어를 배운다면 영어를 사용하는 사람들의 사고 구조가 몸에 익도록 해야한다. 결국 프로그래밍 언어를 배우는 것도 쿵후(工夫)와 크게 다르지 않다. 몸의 길을 닦는 것이다. 태권도라는 것은 결국 손과 발이 갈 '길'이면서 동시에 자신의 몸에 그런 '길'을 내주는 공부다. 어떤 산 근처에 촌락이 형성되면 사람들은 그 산을 다니기 시작할 것이고 '적당한 루트'를 따라 길이 자연스럽게 만들어 질 것이다. 사람들이 많이 다니면 자연스럽게 길이 닦이는 것이다. 한번 길이 놓인 이후로는 별 고생 없이 그 길을 따라가면 된다. 이제는 그 산과 길이라는 것이 별개가 아니고, 또한 그 길과 내 몸도 별개가 아니다. 내 몸 역시 그 길에 맞게 '진화'했을 테니까.

과학자, 장인, 예술가

앞서 설명한 표현 양식과 언어와 사고간의 관계 등을 고려해 볼 때 다양한 패러다임의 언어를 안다는 것(비슷비슷하거나, 자체의 일관된 철학 없이 수많은 기능만 끌어 모은 소위 '강력한' 언어는 별 도움이 안된다)은 문제 해결자들에게 엄청난 무기가 될 수 있다. 이것이 필자가 말하고자 하는 '장인(artisan)으로서의 프로그래머'다. 자신의 몸의 훈련을 통해 도구를 마치 몸의 연장(延長)처럼 만들되, 하나의 도구에만 종속되지 않으며 재료에 알맞게 다양한 도구를 선택할 수 있어야 한다. 또한 자신이 만드는 프로그램 하나 하나에 누가 감시하든 말든 성실하게 최선을 다하고 그것을 내 몸과 같이 아끼고 사랑할 수 있어야 비로소 '장인으로서의 프로그래머'가 될 수 있다.

이와 동시에 프로그래머는 예술가(artist)의 역할도 할 수 있어야 한다. 다익스트라는 "아름다움이 우리가 하는 일이오(Beauty is our business)"라는 명언을 남겼고, 크누쓰 교수의 기념비적 저서 「TAOCP」에도 예술(art)이라는 말이 들어간다{주: 영어에서 art는 기술이라는 의미로 쓰이기도 하지만 크누쓰는 분명히 미적 가치를 추구하는 예술임을 밝히고 있다}. 과학우선주의 사회에서는 이런 예술이나 기술과 같은 것은 저열한 것으로 치부되고,

프로그래머가 알아야 할 것

필자는 이 글에서 '프로그래머'라는 단어를 꽤 폭넓은 의미(소프트웨어 전문가, software professional)로 재정의해 사용했지만, 실제로는 그렇지 못하고 그 역할이 프로그램 개발 전체 과정의 극히 일부분, 즉 소프트웨어 구축에 머무는 경우가 많고, 또 개발자 스스로도 자신이 해야 하고 알아야 할 것은 그것이 전부라고 여기는 것을 많이 볼 수 있다. 더 나아가 어떤 이들은 자바나 C++, 마이크로소프트 윈도우 NT 등과 같은 구체적인 지식이 자신이 알아야 할 모든 것이고 그것을 알면 전문가가 될 수 있다고 생각하는 경우도 있는 것 같다.

한 사람이 소프트웨어를 개발하기 위해 알아야 하는 모든 지식의 절반이 약 삼 년이면 모두 퇴물이 된다고들 한다. 하지만, 어떤 분야가 좀 더 발전하고 성숙해지면서 그런 종류의 지식들과 동시에 수십 년간 변하지 않는 원칙과 원리도 늘어난다. 이런 원칙은 구체적인 기술이 바뀌어도 늘 적용된다. 「No Silver Bullets-Essence and Accident in Software Engineering」을 쓴 프레드 브룩스는 본질(essence)과 비본질(accident)적 성질을 구분하고 있다. 양자는 서양철학의 용어로, 예컨대 자동차에서 기어 변속이 자동이라거나 혹은 수동, 차의 색깔이 어떤 색이라는 것, 타이어의 폭과 같은 것은 비교적 '우연적이고 비본질적'인 성질이라는 것이다. 반면 자동차가 되기 위해 반드시 갖춰야 할 성질을 '본질적 성질'이라고 할 수 있다. 엔진을 갖는다든지 하는 것들 말이다.

그렇다면 소프트웨어 개발에 있어 비교적 오래 지속될 본질적인 '핵심' 지식은 어떤 것일까. 이런 집합을 정의할 수 있을까.

ACM과 IEEE를 비롯한 학계와 산업계의 수많은 참여자들과 함께, UQAM(Université du Quebec à Montreal)이라는 대학에서 주도적으로 진행하는 연구가 있는데, 여기선 소프트웨어 엔지니어링에서 일반적으로 받아들여지는 소위 핵심 지식을 찾는 작업을 하고 있다. SWEBOK(Soft ware Engineering Body of Knowledge, http://www.swebok.org)이라고 하는 이 프로젝트는 한마디로 소프트웨어 전문가가 '알아야만 할' 내용을 찾는 노력을 하고 있는 것이다.

SWEBOK에서 정한 핵심 지식을 크게 보면, 소프트웨어 설계(Design), 구축(Construction), 테스팅(Testing), 형상 관리(Configuration), 품질 공학(Quality Engineering), 프로세스(Process) 관리 등이 있다(이 큰 줄기에서 좀더 구체적인 부분들로 세분화된다). 여기서 프로그램을 구체적으로 설계하고, 코딩하고, 디버깅을 하며, 성능 최적화을 수행하는 것 등은 개발자가 자신이 하는 모든 일이란 거의 소프트웨어 구축이라는 한 가지 지식일 뿐이다. 혹시 자신이 해야 하는 일이 아니라고 생각하는 사람에게는 스티브 맥코넬이 말한 "소프트웨어 전문가라면 모든 분야에 대해 최소한 개론적 지식을 갖춰야 할 것이고, 대부분에 대해 어느 정도의 능력을 가져야 하며, 이 중 일부에 대해서는 거의 통달할 수 있어야 한다"를 전하고 싶다.

그런데 이쯤 되면 이런 것들의 실효용에 대해 의문을 제기하는 사람이 있을 것이다. 작년 IEEE Computer 저널에 흥미로운 연구 결과가 발표됐다(What Knowledge Is Important to a Software Professinal?, Lethbridge, Timothy C., IEEE Computer, May 2000). SWEBOK과 대학의 커리큘럼에서 75개의 주제를 선택해 각각에 대해, 북미지역 및 유럽의 소프트웨어 전문가(평균 10년 정도 종사한 사람들로 프로젝트 관리자나 프로그래머들) 200여명에게 몇 가지 질문을 하고 그 결과를 분석했다.

이 분야의 전문가라고 할 만한 사람들이 생각하는 가장 중요한 주제는 무엇이었을까. 특정 지식이 주는 유용성과 자신의 사고에의 영향 등의 측면에서 중요성을 평가했는데, 중요 항목 20개를 뽑으면 다음과 같다(주: 어떤 분야의 전문가라고 자처하는 사람이 본인이 생각하는 중요한 지식, 기술을 얘기하는 것은 개인체험의 한계를 가질 수밖에 없다. 따라서 이런 통계적인 연구를 통해 검증된 자료가 중요한 가치를 갖는다).

1. 특정 프로그래밍 언어들
2. 자료 구조
3. 소프트웨어 디자인과 패턴
4. 소프트웨어 아키텍처
5. 요구사항 수집 및 분석
6. HCI(Human Computer Interaction) 및 유저 인터페이스
7. 객체 지향 기술 및 개념
8. 직업윤리와 프로페셔널리즘
9. 분석 및 디자인 방법
10. 청중에게 프레젠테이션하는 것
11. 프로젝트 관리
12. 테스팅, 검증 및 QA(Quality Assurance)
13. 알고리즘 설계
14. 기술적 글쓰기(Technical Writing)
15. 운영체제
16. 데이터베이스
17. 리더쉽
18. 형상 및 릴리즈 관리
19. 데이터 전송과 네트워크
20. 경영

대학을 졸업하고 실제 일을 하면서 중요한 것으로(학교에서 가르쳐 주지 않았지만 실전에서 중요한 것) 소프트웨어 디자인과 패턴, 소프트웨어 아키텍처, 요구사항 분석, HCI 및 유저 인터페이스, 프로젝트 관리, 테스팅 및 QA, 형상관리 등을 꼽았다. 그리고 현재 업계에 종사하면서 중요하다고 여겨지는 것 중 자신의 지식이 부족한 것으로는 협상(Negotiation), 리더십, 유저 인터페이스, 요구분석, 직업윤리와 프로페셔널리즘 등을 뽑았다. 이 조사에서 컴퓨터 관련 전공자들은 62%(아주 폭넓은 범위의 참가자에 의해 조사가 수행됐기에 이 비율이 IT 관련 업계에서 전공자들의 평균 비율과 매우 근접하다)였는데, 비전공자들이 일을 해오면서 '전문가'가 되기 위해 자신들이 가장 많이 배워야 했던 것은 무엇이었을까. 특정 프로그래밍 언어들, 자료 구조, 테스팅, 검증 및 QA, 운영체제, 소프트웨어 디자인 및 패턴, 객체지향 기술 및 개념, 데이터베이스, 형상관리, 직업윤리 및 프로페셔널리즘 등이었다.

전문가가 되길 꿈꾸는 학생이나, 이미 업계에 종사중인 실무자들은 이런 전체 그림을 항상 염두에 두고 자신의 지식을 관리하며, 구체적인 지식과 동시에 일반적이고 좀더 오래 지속할 '핵심'을 함께 구유하는 데 힘써야 할 것이다.

그것을 과학화해야 비로소 자랑스러워하는 경향이 강하다. 하지만 아인슈타인의 말대로 "수학의 법칙들이 현실을 언급하는 이상, 그것은 확실하지 않고 그것이 확실한 이상, 그것은 현실을 언급하지 않는다"는 사실을 기억해야 할 것이다. 프로그래밍이라는 것은 자연과학이 될 수 없다. 인간이라는 요소가 개입되기 때문이다. 결국 크누쓰가 말하는 것처럼 우리는 '아름다운 프로그램(beautiful program)'을 만드는 심미적 목적을 가질 수밖에 없는 것이다.

물론 과학자로서의 프로그래머를 무시할 수도 없다. 비록 이론이 모두 적중하지 못하더라도 탄탄한 이론적 바탕은 우리의 수많은 시행착오를 절약할 수 있게 해준다. 분석적이고 논리적인 냉철한 사고를 갖고, 예술가로서의 감수성과 심미성을 추구하며, 자신의 몸 공부가 경지에 이른 장인이 바로 이 시대가 원하는 프로그래머일 것이요, 이 프로그래머가 역시 진정한 엔지니어의 요구사항을 충분히 만족시킬 수 있을 것이다.

훌륭한 프로그램과 프로그래머

우리 모두는 훌륭한 프로그래머이기를 꿈꾼다(어쩌면 우리의 관리자들이 더욱 그러할 수도 있겠다). 훌륭한 프로그래머는 어떤 사람일까. 프로그래머는 프로그램으로 말한다. 이들은 오로지 프로그램(결과)과 프로그래밍(과정)으로 평가받기를 원한다. 그렇다면 훌륭한 프로그래머란 바람직한 프로그래밍을 통해 훌륭한 프로그램을 만들어 내는 사람일 것이다. 오로지 결과물, '프로그램'만 좋으면 된다고 생각하는 사람은 이 사회에 적응하기 힘든 해커 기질을 타고났다고 자신해도 좋을 듯 싶다. 프로그램이라는 결과물에는 그 사람이 작업한 과정은 잘 드러나지 않는다. 결과물이 아무리 중요하다고 해도 하루 걸릴 일이 일년 걸리거나, 천원에 할 일을 일억원에 하는 경우를 생각해 보라.

하지만 '좋은 프로그램'이 '좋은 프로그래머'를 결정하는 데 가장 중요한 요인이라는 것을 무시할 수는 없겠다. 어찌 보면 좋은 프로그래밍을 좋은 프로그램의 필요조건으로 넣을 수도 있겠는데, 그렇다면 결국 '좋은 프로그램'이 지상최대의 궁극적 과제가 될 수도 있을 것이다(주: 사실 이런 논의는 그 자체로 무의미할 수 있다. 좋은 프로그램이라는 것은 맥락 의존적이기 때문이다. 어떤 상황에서 좋은 프로그램이 다른 곳에서는 나쁜 프로그램이 될 수 있다. 하지만 우리는 비교적 일반적인 경우에서 공통분모를 놓고 생각해 보려고 한다). 좋은 프로그램이 되기 위한 조건들은 다음과 같다.

- ◆ 스펙(Specification)
- ◆ 스케쥴
- ◆ 적응성
- ◆ 효율성
- ◆ 경제성

「The Pragmatic Programmer」의 저자 데이비드 토마스와 앤드류 헌트는 프로그래밍이란, 단순히 보면 자신이 혹은 사용자가 원하는 일을 컴퓨터가 할 수 있도록 만드는 것이라고 했다. 무엇이 좋다 나쁘다, 훌륭하다 못하다는 효용가치 판단은 구체적 목적이 있어야 비로소 가능해진다. 분명 프로그래밍의 목적은 '문제 해결'이요, '일을 해 내는 것'이다. 그렇다면 좋은 프로그램의 가장 기본적인 요구사항은 '해야 할 일을 하는 것'이라고 말할 수 있을 것이다. 이것이 첫 번째이자 가장 중요한 '스펙'이라는 사항이다. 즉 프로그램은 모든 요구조건을 만족해야 한다. 이것은 효율성(속도 등)과는 비교할 수도 없을 만큼 중요한 사항이지만 많은 개발자들이 이것을 망각한 채 다른 잔가지에 매진하곤 한다.

두 번째 스케쥴은 '시간 안에' 프로그램을 만들어 내는 것을 말한다. 요즘 같이 소프트웨어 산업에서 비즈니스 측면이 부각되는 때라면 이 요건은 더욱 중요해진다. 물론 이 스케쥴의 요건을 충족하기 위해서는 적절한 개산(槪算, estimation)이 필요하다.

한 사회의 안정성은 그 사회의 가변성에 의존한다. 자기 모순적 명제 같지만, 우리는 이러한 사실을 실생활에서 자주 접한다. 프로그램의 경우를 보자. 가변성이 낮은 프로그램일수록 일찍 폐기 처분되기 쉽다. 이런 프로그램은 다른 환경(유사하지만 조금 다른 문제 또는 다른 플랫폼 등)에 잘 적응하지 못하고, 그로 인해 비용이 극도로 늘어나게 된다. 고정화, 고형화는 죽음이고, 유연성은 삶이다. 유전학과 자연선택에서 유명한 피셔(R. A. Fisher)는 '피셔의 기초 원리'를 만들었다. 골자는 "특정 환경에 잘 적응된 시스템일수록 그것은 새로운 환경에 잘 적응하지 못한다"는 것이다. 어떤 프로그램이 효율적으로 되기 위해서는 현 문제의 특수성과 그 프로그램이 실행될 컴퓨터의 고유한 특성을 '영리하게' 이용해야 한다. 대부분의 경우 특출한 기술을 사용해 번 시간은 그 코드에 새로운 부분이 추가될 때 소모되는 시간에 비해 극히 미미하다. 배보다 배꼽이 큰 셈이다. 현재는 하드웨어의 비약적인 발전으로 과거 만큼 빡빡한 한계는 없다(물론 상대적으로 다뤄야 할 문제가 커진 것에서 생기는 한계상황도 있다). 컴퓨터가 이해할 수 있는 코드를 만드는 것이 문제가 아니다. 문제는 인간이 쉽게 이해할 수 있는 코드이고, 자신의 변화를 허용하는 코드인 것이다.

효율성은 실행 시간만을 의미하는 것은 아니다. 실행 시간이라는 것은 실행 이전이나 이후에 소요되는 시간으로 어느 정도 대치할 수 있기 때문이다. 또한 효율성의 측정이라는 것을 진지하게 생각해 봐야 한다.

짧은 평균 실행 시간보다 낮은 표준 편차가 더욱 중요할 수도 있다. 입력자료가 100개 미만일 때 1밀리세컨드 걸리지만 그 이상에선 10초 이상 걸리는 경우는 표준 편차가 너무 크다. 그리고 효율성 증진을 위해서는 여러 가지 최적화의 방법이 존재하지만 '포기에 의한 최적화'도 가능하다. 일반적으로 어떤 문제가 풀 수 있는 영역에서 비교적 덜 중요한 10%를 포기하게 되면 전체 실행 효율(속도)을 두 배 이상 올릴 수 있다. 이런 경우에는 '스펙'을 수정해야 한다(이 때 고객 또는 자신과의 솔직한 커뮤니케이션이 꼭 필요하다). 어찌 됐건, 효율성(efficiency)이라는 가치는 '바른 일을 하는가'라는 효과성(effectiveness)의 고려 없이는 전혀 무의미하다.

마지막으로 경제성은 비용과 수익의 측면이다. 프로그래밍의 목적이 '돈을 버는 것'이라고 할 때 가장 우선적인 요소가 될지도 모르겠다. 하지만 경제성 역시 다른 요건이 충족될 때 고려해 볼 수 있다.

이상의 요건들을 '상황에 맞게' 적절히 갖춘 프로그램을 생산해 낼 수 있는 사람, 우리는 그들을 '훌륭한 프로그래머'라고 부른다(주: 여기서 언급되지 않았지만 훌륭한 팀 플레이어의 역할도 빼놓을 수 없다. 뛰어난 지식이 있어도 의사소통 능력이 떨어지는 사람이 컴퓨터 도사의 전형이 되던 시대는 갔다).

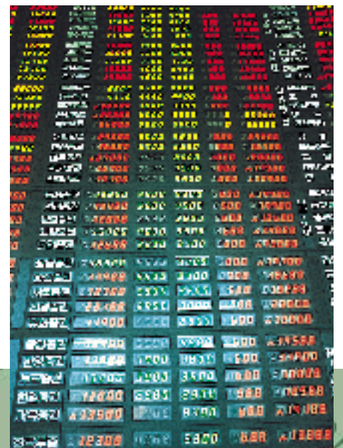
프로그래머의 지식 투자

수많은 세계의 석학은 미래사회는 지식 중심 사회가 될 것이라고 예견했고, 그 예측은 거의 맞아 들어가고 있는 것 같다. 몇몇 특수한 상황을 제외하곤, 지금 당장 가장 수익률이 좋고 안정성이 보장되는 투자는 지식에 대한 투자다. 한 번 제대로 획득하면 잃어버릴 일이 거의 전무하기 때문이다. 프로그래머와 같은 지식 산업 종사자들은 (자신들은 단순 노무직이라고 부인할 지라도) 자신의 지식 투자를 효과적, 효율적으로 관리할 수 있어야 한다. 이것이 미래에 대비하는 길이다.

앞서 소개한 헌트와 토마스는 이것을 지식 포트폴리오(Knowledge Portfolio)로 설명하고 있다. 지식 투자 관리는 어떤 면에서 재무 투자 관리와 상당히 유사하다.

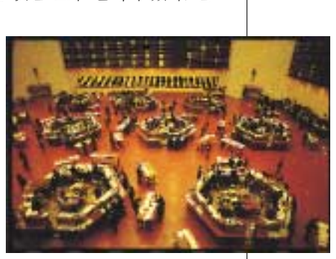
1 정기적으로 투자하는 것을 습관화한다.
2 분산 투자가 장기 성공의 열쇠다.
3 자신의 포트폴리오를 안전한 것과 위험성이 큰 것을 고루 갖추도록 균형을 잡는다.
4 싸게 사서 비싸게 판다.
5 주기적으로 포트폴리오를 재평가한다.

첫 번째, 정기적 투자는 자신의 지식을 위해 '정기적'으로 '조금씩' 새로운 지식을 습득하는 것을 말한다. 예를 들어 매년 새로운 프로그래밍 언어(가능하면 다른 패러다임의 언어)의 습득을 목표로 세운다든지, 매 분기마다 새로운 기술 서적을 한 권씩 읽기로 약속한다든지, 비기술 서적(컴퓨터와 관련 없어 보이는 것일수록 좋다)을 정기적으로 읽고 통신 동호회에 가입해 스터디를 한다든지, 새로운 플랫폼을 시도해 보거나, 매달 잡지와 저널을 구독하는 노력을 꾸준히 조금씩 해주는 것이 중요하다. 지금 당장 필요하지 않은 또는 향후 수십 년간 사용할 것 같지도 않은 언어를 배우면서 자꾸 의혹 속에 괴로워하지 마라. 설령 그 언어를 사용하지 않더라도 다른 문제 해결에 분명 도움이 될 것이다. 또 경계를 넘어선 '가로지르기'도 필요하다. 갑이라는 기술을 배웠고 전혀 다른 분야에서 을이라는 지식을 얻었을 경우 둘 사이에 존재하는 상관관계, 응용 방법 등을 깊이 생각해 보는 것은 지식을 체화하면서 실질적인 소득도 얻을 수 있는 기회다.



두 번째, 분산 투자는 한마디로 서로 다른 것을 더 알게 될수록 자신은 더욱 가치 있어질 것이라는 뜻이다. 하지만 기본적으로 어떤 것이 저무는 태양이고, 어떤 것이 떠오르는 것인지에 대한 정보를 갖춰야 할 것이다. 그렇다고 자신을 너무 내몰지는 마라. 어제 떠오른 것이 오늘 지고, 어제 졌던 것이 오늘 뜨는 현상이 이 분야에 한 두 번 있는 일이던가.

세 번째, 위험 관리다. 아마도 '달걀을 한 곳에 담지 마라'는 말을 들어봤을 것이다. 안정성이 보장되는 지식만 좇는 것이나, 위험성이 너무 큰 (실효가 입증되지 않은) 지식만 추구하는 것은 모두 문제가 있다. 균형을 유지해야 한다.



네 번째, 싸게 사서 비싸게 팔아 차익을 크게 만들어라. 자바가 처음 등장했을 때 그것을 배우는 일은 상당히 위험성이 높았을 것이다. 하지만 초기 개발자들이 결국은 얼마나 비싸게 자신의 지식을 팔아 이득을 얻었는가. 서점에 널린 기술은 이미 비싸게 사서 싸게 팔 확률이 높은 것들이다.

다섯 번째, 이 분야는 매우 동적이다. 지난달에 인기가 치솟았던 것이 이번 달엔 곤두박질칠 수도 있다. 너무 유행에 민감한 것도 문제겠지만, 그렇다고 자신의 포트폴리오를 몇 달 간 다시 들여다보지 않는 것처럼 위험한 것도 없다. 먼지가 쌓이고 녹슬어 가는 지식이라면 기름칠을 해야 할 것이고, 새로운 지식이 필요하다고 판단되면 과감이 뛰어들어야 한다.

이 다섯 가지 중에서 뭐니뭐니 해도 가장 중요한 것은 첫 번째 항목인 '자신의 포트폴리오에의 정기적인 투자'가 될 것이다.

프로그래머의 자기 수련

서적

다음의 서적들은 프레드 브룩스, 팀 버너스 리, 앨런 쿠퍼, 제임스 고슬링, 브라이언 커니건, 스티브 맥코넬, 앤드류 타넨바움, 윌리엄 스톨링, 제럴드 웨인버그 등과 같은 컴퓨터 역사에 이름이 남을 만한 유명인으로부터 실전에서 20년 이상을 구르고 베테랑으로 알려진 프로그래밍의 노장과 달인들이 공통적으로 '자신에게 가장 많은 영향을 준 컴퓨터 관련 책'으로 꼽은 것을 필자가 몇 년에 걸쳐 수집하고 추려낸 것이다. 재밌게도 컴퓨터와는 별 관련 없어 보이는 책들도 몇 권 있다. 역시 유행과 동떨어졌기에 유행에서 살아남을 수 있었던 것이리라. 여기 나열된 책들은 대부분 처음 출판된 지 10년이 넘은 것이고 어떤 것은 30년이 넘은 것도 있다. 이런 고생대의 화석이 아직까지도 우리 시대에 유효할 수 있다는 것 자체가 경이로울 뿐이다. 정말 좋은 책이란 읽을 때마다 새로운 맛이 소록소록 나오고, 자신이 가진 문제에 늘 다양한 해답을 제공해 주는 것이다. 수천 년을 면면히 이어 내려온 노자 도덕경이 인류 곁을 아직 떠나지 않는 이유일지도 모르겠다. 전문 프로그래머라면 항상 기술 중심적이고 구체적인 책을 한 손에 들고 공부하면서도, 이런 일반적이고 유행과 상관없는 책을 다른 손에서 놓치지 않아야 할 것이다.

◆ The Art Of Computer Programming (TAOCP), Knuth, Donald

알고리즘과 자료구조에 관한 최고의 책이다. 프로그래머로서 정말 충실하게 공부해 둬야만 나중에 좌절을 맛보는 경험을 피할 수 있다. 현재 세 권까지 출판돼 있고, 1997년에 세 번째 판이 나왔다. 읽을 자신이 없다면 최소한 이 책들의 목차만이라도 봐두자. 상대적으로 좀 가벼운 책으로는 로버트 세드게윅(Robert Sedgewick)이나 토마스 코멘 외 2인 공저의 'Introduction to Algorithms'를 참고하라.

◆ Programming Pearls, Bentley, Jon Louis

실질적인 코드(C, C++)와 함께 알고리즘 개선, 코드 최적화 등을 다룬다. CACM에 연재됐던 것을 모으고 좀더 덧붙인 것이다. 현재 프로젝트에서 알고리즘이나 자료구조에서 문제가 생기면 일단 마음을 차분히 가라앉히고 조용한 곳에서 이 책을 읽어보라.

◆ Structure and Interpretation of Computer Programs, Abelson, Harold, et al.

미국 MIT 대학에서 10년이 넘도록 입문 코스용 교과서로 사용되고 있는 유명한 고전이다. 비록 수년이 흘렀고, Scheme이라는 그다지 대중적이지 못한 언어를 사용했지만, 이 책은 여전히 고전으로서 가치가 빛나고 있다. 세월이 가도 변치 않을 프로그래밍의 근본 원리 전달을 목적으로 집필됐기 때문일 것이다. 이 책은 겉 표지에 마법사 그림이 있어서 마법사 책이라고 불리며, 에릭 레이먼드의 해커 사전에도 등재돼 있다.

◆ Design Patterns, Gamma, Erich, et al.

하나의 디자인 패턴은 특정한 종류의 문제를 해결하는, 프로그래밍 언어보다는 좀더 추상적인 차원에서 일반적인 방법을 서술한다. 저자들은 '네 명의 동지들(Gang of Four)'로 더 알려져 있다. 국내 서점에서도 바닥이 날 정도로 잘 팔리는 베스트 셀러다.

◆ A Pattern Language: Towns, Buildings, Construction, Alexander, Christopher

패턴 언어는 원래 건축학에서 온 개념이다. 건물을 짓는 것과 소프트웨어를 만드는 것(영어로는 모두 build라고 한다) 간에는 상당한 유사점이 있다. 디자인 패턴을 본 사람이라면 이 개념의 원류를 공부해 보는 것이 매우 유익할 것이다. 이 책과 함께 많이 읽히는 'Timeless Way of Building'은 좀더 철학적(노장사상과 관계가 깊다)이고 사변적이다. 이것을 읽는다면 세상을 보는 관점이 바뀔 것이다. 흔히 알렉산더의 이론에 대한 반대로 실증적인 결과와 예가 없다는 것인데, 그의 'The Production of Houses'를 꼭 읽어보길 권한다. 필자는 이 책에서 프로젝트 관리와 적응적 개발(adaptive development) 등의 가능성을 발견했다.

◆ How Buildings Learn: What Happens After They're Built, Brand, Stewart

비교적 최근에 출간된 책으로 건축학적인 개념에서 어떻게 건물이 '진화'하고 스스로 변화시켜 나가는지를 보여주며, 진화하기 좋은 건축물은 어떤 것인가에 대한 진지한 고찰이 들어있다. 우리 프로그래머들이 고민하는 문제와 동일하다. 따라서 프로그래밍을 새로운 관점에서 볼 수 있을 것이다.

◆ The Mythical Man-Month: Essays on Software Engineering, Brooks, Frederick

더 이상 설명할 필요가 없는 책이다. 진행 중인 프로젝트 팀에 더 많은 인원을 쏟아 부으면 오히려 제품 출시가 더욱 늦어진다는 점을 밝힌 것으로 유명하다. 소프트웨어 공학에 관심이 없거나 기반 지식이 전무한 사람도 읽어볼 만한 책이다.

◆ Code Complete, McConnell, Steve

소프트웨어 구축 과정에 관한 한 거의 모든 사항을 '코드 중심으로' 모아둔 집적체다. 필자는 아직까지 이 주제를 다루면서 이 정도로 포괄적이면서 동시에 가치있는 책을 보지 못했다. 특히 33장의 참고자료는 더 많은 자료를 원하는 사람에게 매우 유용하다.

◆ How to Solve It, Polya, George

문제 해결에 관한 한 최고의 베스트 셀러다. 한글 번역판이 있는데, 국내에서 이 책의 가치가 제대로 평가되지 못하고 수학 시험 준비 서적으로 분류되고 있는 점이 아쉽다. 비단 수학뿐만 아니라 거의 모든 '문제 해결' 이라고 할 만한 것(프로그래밍을 포함)에 대해 건강한 경험적 가이드라인(휴리스틱스, heuristics)을 제시하는 책으로 교육적 가치도 높다. 이 책을 공부하고('읽고'가 아니라) 나면 한층 똑똑해진 자신을 발견할 수 있다.

◆ Godel, Escher, Bach: An Eternal Golden Braid, Hofstadter, Dougl as R.

GEB라고도 불리는 이 책은, 저자 호프슈테더 교수에게 풀리처상을 안겨줬다. 아마 컴퓨터 관련 직종뿐만 아니라 자연과학, 철학 쪽에서까지 널리 읽히는 인기 서적이 아닐까 한다. 수학의 괴델과, 회화의 에셔, 음악의 바흐 작품을 비교하며 공통점을 찾는다. 전산학의 시원이라 할 수 있는 튜링 컴퓨터에 대한 설명이 있다.

◆ Computer Architecture: A Quantitative Approach, Patterson, David A., et al.

전문 프로그래머라면 하드웨어적인 지식도 절대 놓쳐서는 안된다. 이러한 지식을 아는 사람과 그렇지 못한 사람의 프로그래밍 능력과 몸값은 엄청난 차이가 있다. 컴퓨터 아키텍처에 관한 한 최고의 양서로 평가받는 이 책은 학부생이나 평범한 프로그래머들이 보기엔 다소 난해할 수 있다. 그럴 경우에 같은 저자의 'Computer Organization and Design' 을 보는 것이 좋다.

◆ Elements of Style, Strunk, William and E.B. White

영미권에서 작문 관련 서적으로 가장 많이 팔린 책이다. 브라이언 커니건과 플로거가 쓴 'The Elements of Programming Style' 은 이 책의 제목을 흉내낸 것이다. 영미인과 문법이나 철자법 등에 대한 논쟁을 하다가도 "스트렁크와 화이트의 책에 따르면"이라는 한마디면 종지부를 맺을 수 있을 정도로 권위적인 책이다. 특히 5장 스타일에 대한 가이드는 기술적 문서를 작성할 때는 물론이고 프로그래밍을 할 때에도 참고가 될 것이다.

◆ The Psychology of Computer Programming, Weinberg, Gerald M.

이 책은 에릭 레이먼드가 썼던 '성당과 시장' 에서 비자아적 프로그래머(egoless programmer)에 관한 언급으로 국내에 많이 알려졌지만, 사실 영미권에서는 이미 베스트 셀러의 반열에 오른 지 몇 십 년이 됐다. 프로그래밍을 인간활동의 하나로 인식하고 심리학적인 접근을 통해 새로운 분야를 세운 기념비적인 책이다. 아직까지도 ACM이나 IEEE 회원들이 가장 많이 구입하는 책 중 하나이며, 최근 실버 기념판이 출판됐다.

◆ ACM Turing Award Lectures : The First Twenty Years : 1966 to 1985

튜링상은 컴퓨터 분야의 노벨상이다. 우리가 실제로 사용하고 있는 거의 모든 기술의 원천은 튜링상 수상자들의 작품이다. 이 책은 튜링상 수상시 함께 하도록 돼 있는 강의 내용을 20년간 모은 것이다. 한눈에 컴퓨터계의 발전 역사를 조망할 수 있으며, 선지자들이 조심스럽게 말하는 앞으로의 발전 방향도 엿볼 수 있다. 특별히 엣저 다익스트라(Edsgar Dijkstra)의 'The Humble Programmer' 와 도널드 크누쓰의 'Computer Programming as an Art' 는 꼭 읽어볼 만하다. 각각 72년, 74년에 한 강의이지만, 많은 부분이 오늘날에도 유효하다는 사실이 그 페이퍼의 질을 보장해 준다.

소개한 책들은 이미 십년 이상을 살아 남았고 앞으로도 최소 오년 이상 가치를 유지할(혹은 더 높아질) 책이 대부분이다. 그런데 모아 놓고 보니 모두가 원서다. 참 슬픈 일이다. 아직까지 우리나라에 그 가치가 최소 오년 이상 되는 책이 쓰여지고 있지 않다는 것은 짧은 역사와 기술 도입만으로는 설명하기 힘들다. 게다가 필자가 아는 한 그나마 두 권(GEB, How to Solve It)을 빼놓고는 모조리 번역 작업조차 되지 않았다. 멀리 보지 못하고, 멀리 볼 겨를도 없는 우리 신세가 안타깝지만, 조만간 이런 작업들이 진행되리라는 일말의 희망을 걸어본다.

정기 간행물

필자는 과거를 보려면 서점에 나가보고, 현재를 보려면 정기 간행물이나 논문을 살펴보고, 미래를 보려면 현장(아카데미아와 기업계)을 뛰어다녀 봐야 한다고 생각한다. 이미 서점에 판을 치고 있는 기술들은 그 정보 가치가 많이 하락한 것들이고 소위 '끝물' 일 확률이 높다. 주식의 "소문에 사고 뉴스에 팔아라"는 말이 적용되는 것이다. 잡지 같은 것은 비교적 출간 사이클이 짧고, 이에 따라 현실 세계를 조금 더 빨리 반영한다. 국내에도 물론 좋은 잡지와 저널이 많이 있지만 여기서는 언급을 피하도록 하겠다.

◆ Software Development (www. sdmagazine.com)

이 잡지는 특정 기술이나 팁보다는 전문 프로그래머가 접하는 일반적인 프로그래밍 관련 이슈를 다룬다. 소프트웨어 공학, 개발 방법론, 프로그래머란 직업에 대한 기사들 혹은 프로젝트 관리자에게 도움이 될만한 것들이 많고, 개발 툴 리뷰도 유익하다. 특히 일년에 한번씩 있는 졸트상 수상은 꼭 놓치지 말아야 할 좋은 정보다.

◆ Dr. Dobb's Journal(www.ddj.com)

제목에는 저널이라고 돼 있지만 그다지 아카데믹한 내용은 아니다. 비교적 이론적이고 코드 지향적인 성격의 잡지라고 보면 된다. 알고리즘 분야에 상당히 강하다. 도구나 기술 사용법의 연마도 중요하지만 도구와 기술의 이면에 있는 이론으로 중무장하는 것은 '전문 프로그래머' 가 되기 위해선 반드시 해야할 일이다.

◆ Information Week (www.inform ationweek.com)

아무리 개발자라고 해도 비즈니스적인 변화는 늘 감지하고 있는 것이 좋다. 어떤 기술이 잘 팔리고 향후 어떤 기술이 주목을 받을지, 지금 내가 의지하고 있는 배가 침몰 중이지는 않은지 등을 말이다.

◆ Communications of ACM (www. acm.org)

가장 오래되고, 또 가장 인정(?)받는 컴퓨터 관련 저널 중 하나. IEEE Computer처럼 다루는 분야가 매우 넓기 때문에 대부분의 기사가 자신의 관심사 밖의 것일 수도 있는데, 컴퓨터 관련 이론과 기술의 원류가 되는 만큼 자신의 분야를 막론하고 꼭 구독해 볼 가치가 있는 잡지이다. 이 곳에 기사가 실리고 몇 년 지나야 관련 내용을 여타 잡지에서 확인하는 경우도 종종 있다.

◆ IEEE Software(www.computer.org)

격월 잡지로 소프트웨어 개발 방법론이나 여타 소프트웨어 관련 주제를 다룬다. 전문 프로그래머들에게 도움이 될 만한 실질적인 기사가 많이 있다. IEEE Computer는 앞서 소개한 CACM과 비슷한 성격인데, IEEE Software보다는 다루는 주제의 범위가 훨씬 넓다.

모임

컴퓨터 분야에는 현재 두 개의 '세계 수준의' 전문가 모임이 존재한다. ACM(Association for Computing Machinery)과 IEEE 컴퓨터 소사이어티가 그것이다. (세계 수준의) 전문 프로그래머라면 최소한 둘 중 하나에는 가입되어 있는 것이 일반적이다. 국내에도 다양한 전문가 단체가 있는데, 자신의 관심 분야 한 곳과 일반적인 한 곳은 꼭 가입을 해두는 것이 좋다. 또한, 통신 동호회나 스터디 그룹과 같은 비격식적인 모임에도 참여하고, 자신과 다른 분야에서 일을 하는 전문가들과 지속적인 접촉을 유지하는 것이 세상에 뒤쳐지지 않는 방법이다. 항상 다른 사람들이 '현재 고민하는 것' 이 무엇이고, '성취하고자 하는 것' 이 무엇이며, '과거에 어떤 교훈을 얻었는지' 의 세 가지를 탐지할 일이다.

정리 : 박은정 whoami@sbmedia.co.kr

효과적인 공부법

심리학에서 스키너 상자는 행동주의(Behaviorism)의 대표적인 실험인데, 상자 안에 생쥐를 가둬두고 그 놈이 버튼 혹은 레버를 누를 때마다 구멍으로 먹이를 보내준다. 처음에는 그 버튼과 레버가 무슨 의미인지를 알지 못하다가 우연히 그것을 누르고는 먹이를 먹는다. 물론 당시에는 그 두 가지 사건을 연관지어 생각하지 못한다. 하지만 시간이 갈수록 학습을 하게 되고, 배가 고플 때면 버튼을 알아서 누른다.

인간의 학습에 대한 이런 스키너 상자식 접근은 이미 폐기 처분된 지 오래지만, 여전히 유효한 것들이 몇 가지 있다.

이 실험에서 버튼이 눌려진 때와 먹이를 보내주는 때의 시간 차이를 늘려주면 학습이 느려지거나 혹은 아예 학습이 일어나지 않는다. 우리가 학습을 할 때 피드백이라는 것은 아주 중요한 역할을 한다. 내가 무엇을 잘못했는지, 제대로 했는지를 즉각적으로 알 수 있으면 바로 자신의 이전 행동에 대한 수정이 가능하고 이는 새로운 학습으로 이어질 수 있다.

이런 면에서 파이썬이나 스몰토크 같은 대화형 인터프리터 언어는 중요한 위치를 차지한다. 난생 처음 파이썬을 대하는 사람이 단박에 문자열을 출력해 본다든지, 사칙연산을 해본다든지 하는 경험은 다른 컴파일러 언어에서 쉽게 체험할 수 없는 '놀라운 학습' 의 경험인 것이다.

주변에서 보면 처음부터 모든 완벽을 기하려는 사람보다 조금씩 시도해 보고 또 수정하고, 다시 시도하는 소위 점진적 접근법을 행하는 사람들이 더 빨리 학습하는 경우가 많이 있다. 예컨대, HTML을 공부할 때 처음부터 태그를 외우고 프로토콜의 정의를 이해하고, 문법을 외우는 등 '정규 코스' 를 밟은 후에 각종 HTML 에디터를 공부하는 경우와, 나모 같은 소프트웨어를 통해 손쉽게 만들어 본 것을 바로 확인해 보고 또 수정하고 하는 식으로 일단 감을 잡은 사람이 차후에 태그나 문법을 공부하는 경우를 비교해 볼 수 있다. 규칙 중심적 학습은 경험 중심적 학습에 비해 훨씬 더 어렵고 비효율적일 수밖에 없다. 규칙은 '데이터' 가, 실제 경험이 이미 내 안에 있은 이후에나 의미가 있다. 아담이 말을 보고 '말' 이라고 이름 지은 것은 말을 보기 전이 아니라 그 후였음을 생각해 봐야 할 것이다.

자신이 사고하는 것에 대해 사고하는 것, 이 이차적 사고를 메타인지(meta-cognition)라고 한다. 자신의 잘못을 스스로 인식하고 그것에 대해 생각해 보는 것도 훌륭한 학습의 경험이 된다. 예컨대, 25-16을 19라고 대답한 학생과 41이라고 대답한 학생을 생각해 보자. 만약 학생들이 독학하는 것이 아니고 '보통' 선생님의 가르침을 받고 있다고 치면, 그 둘은 똑같이 틀렸다는 말을 듣고 자신의 실수를 잊어버릴 것이다. 하지만 두 번째 학생의 경우 명백히 뺄셈 대신 더하기를 했다는 것을 알 수 있고, 첫 번째 학생의 경우도 뺄셈을 하되 십의 자리에서 뭔가 문제가 있었다는 것을 알 수 있다. 이런 정보를 기반으로 개개인에게 좀더 훌륭한 학습의 기회를 제공할 수 있다. 혼자 프로그래밍을 공부하는 사람이라면 자신의 프로그램이 제대로 돌지 않는 경우, 자신의 잘못을 분석하고 어떤 '사고의 틀' 이 문제를 일으켰는지 생각해 봐야 한다. 한마디로 '자기 자신을 볼 줄 알아야 한다' 는 것이다. 이것이 남보다 빨리 학습하는 비결이다.