nML과 프로그래밍 기술의 발달

Progress in Programming Language System: an nML Perspective

이광근
서울대학교 컴퓨터공학부

요 약

ML로 대표되는 프로그래밍 언어의 발전에 녹아있는 프로그래밍 기술의 발달과정을 살펴본다. 그럼으로써, 많은 사람들이 첨단으로 생각하는 컴퓨터 프로그래밍 기술의 알맹이라는 것이 사실은 아직 영글지 않은 미숙한 단계라는 것을 상기해 보도록 한다. 또, ML의 한국 사투리인 nML을 소개한다.

1 프로그래밍을 지배하는 프로그래밍 언어

생각이 바뀌면 어려웠던 문제가 쉽게 풀린다. 문제를 바라보는 시각이 조정되는 순간에, 예전에 어렵게 보였던 문제가 쉽게 풀리는 것이다.

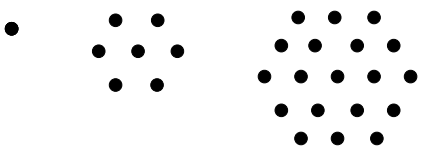
(문제 예1) 50킬로미터 떨어져 있는 자전거 두대가 마주보며 시속 25킬로로 달리기 시작했다. 마주달리는 두대의 자건거 사이를 부지런히 왕복하는 파리가 있다. 파리는 시속 50킬로로 비행한다. 자전거가 부딛힐 때 까지, 파리가 비행한 거리는?

철수의 답) 철수는 수열과 극한, 미분과 적분을 이용한다. 따라서 위의 문제는, 자전거 사이의 폭의 변화, 그 변화하는 폭을 왕복하는 파리의 비행거리를 나타내는 무한 수열의 합으로 해결한다. 수열과 극한, 미분과 적분이 철수가 가진 프로그래밍 언어인 것이다.

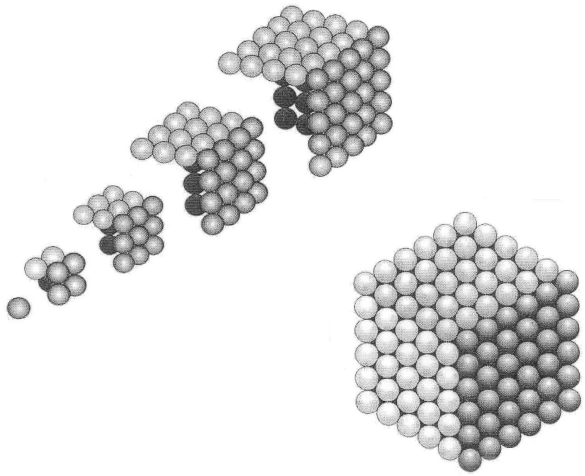
순이의 답) 순이가 가진 프로그래밍 언어는 다르다. 철수 같이 거리를 구체적으로 궁리하는 언어를 사용하지 않는다. 순이의 프로그램은 새로운 각도에서 간단히 고안된다: 파리가 비행하는 거리의 변화라는 복잡한 과정에서 눈을 떼서, 파리가 비행한 총 시간을 계산한다. 그 시간은 자전거 충돌때까지의 시간과 같다. 그 시간은 1시간이고, 따라서 파리의 총 비행거리는 50킬로미터이다. 순이는 문제를 해결하는 데 필요한 만큼의 상위의 수준에서 상황을 정의하고 해결하는 언어를 구사하는 것이다.

(문제 예2) 육각형 정수는 1, 7, 19, 37, 61, 91, 127 등인데, 정육각형 모양이 되도록 바둑판 위에 놓아야 할 바둑알들의 갯수들이다(그림 1). 이러한 육각형 정수들을 처음부터 합해가면 항상 어느정수의 3승이 된다고 한다. 사실인가?

철수의 답) 철수가 구사하는 언어는 n번째 육각형 정수를 정의하는 식을 도출하고, 그 식들을 1번째부터 n번째까지 더해서 만들어내는 정수가 항상 어떤 수의 3승이 된다는 것을



[그림 1] 육각형수



[그림 2] 육각형수는 정육면체의 한꺼풀. 위와같이 3면체의 함 꺼풀씩을 쌓아가면 꽉찬 정육면체를 만들 수 있는데, 각각의 3면체를 오른쪽 중앙의 꼭지점을 중심으로 보면 바로 육각형수(오른쪽 아래) 이다. 참조: *What is Intelligence?*, Jean Khalfa (ed.), Cambridge University Press, 1994

증명한다. 증명은 물론 귀납법이다. 점화식 수열과 그 합, 그리고 귀납법이 철수가 가진 프로그래밍 언어인 것이다.

순이의 답) 순이는 이번에도 철수보다 상위의 수준에서 답을 만드는 언어를 구사한다. 순이는 육각형 정수들의 그림은 정육면체의 한꺼풀에 해당하고, 그 한꺼풀들이 차곡차곡 쌓이면 항상 꽉 찬 정육면체를 만든다는 것을 보인다(그림 2). 따라서, 육각형 정수들을 차례대로 합하면, 정확하게 어떤 정육면체를 꽉채우는 알갱이들의 갯수가(즉, 어떤수의 3승이) 되는 것이다. 2차원과 3차원의 도형이 순이가 사용한 프로그래밍 언어인 것이다.

이렇듯이, 문제를 정의하고 답을 궁리하는 생각의 틀을 적절히 선택할 필요가 있다. 그러면, 만들어지는 해답은 작고 간단해지고 이해하기 쉬워지기 때문이다.

그렇다면, 컴퓨터 프로그래밍의 세계에서 문제를 정의하고 답을 궁리하는 생각의 틀은 무엇으로 결정될까? 다른 많은 것들이 있겠지만, 크게 영향을 미치는 것이 프로그래밍 언어일 것이다.

2 값중심의 프로그래밍

아래의 프로그램을 생각해 보자.

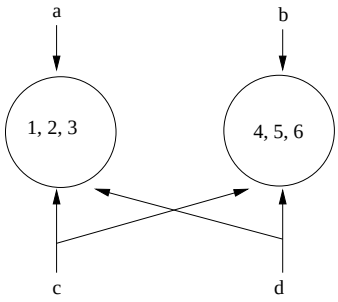
```
a := 1;
b := 2;
c := a + b;
d := a + b;
```

위의 프로그램이 실행되었다고 하고, 몇가지 질문을 던져보자.

– c 의 3 이 d 의 3 인가?

– a 를 바꾸면 c 도 바뀔까?

– d 를 바꾸면 c 도 바뀔까?



[그림 3] "항상 공유하기"방식의 집합 표현

– `e := c` 를 수행하려면 `c` 를 복사해야 하는가?
– `c` 갖고 일 봤으면, 그 3 을 없애도 되는가?

이러한 질문들은 싱겁기 그지없다. 답하기 쉬운 이유는 위의 프로그램에서 다루는 값이 간단한 것(정수)이기 때문이다. `c` 의 3 은 `d` 의 3과 같다. 프로그램 수행후에 `a` 를 바꾼다고 해서 `c`가 바뀌지는 않는다. `c`는 3이라는 값을 계속 가지고 있게된다. 마찬가지로 `d`를 바꾼다고 해서 `c`가 가진 값 3이 바뀌지는 않는다. `e := c`를 수행하면 3이라는 `c`의 값이 `e`에 저장된다. `c` 갖고 일 봤으면, `c`가 가진 값 3은 지워버려도 된다.

이제, 정수보다는 복잡한(컴퓨터에 구현하려할 때 할 일이 많은) 값을 다루는 프로그램에 대해서 비슷한 질문을 해보자. 집합을 다루는 프로그램을 생각하자. 아래 프로그램에서, `set(1,2,3)`은 1,2,3으로 구성된 집합을 만들고, `setUnion(x,y)`는 집합 `x`와 `y`를 합집합을 만든다.

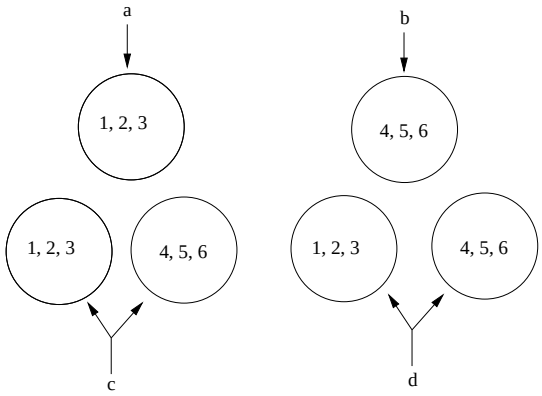
```
a := set(1,2,3);
b := set(4,5,6);
c := setUnion(a,b);
d := setUnion(a,b);
```

위의 프로그램이 실행된 후, 이전과 같은 질문을 해 보자.

– 위의 프로그램에서 `c`의 집합이 `d`의 집합과 같은가?
– `a`의 집합을 바꾸면 `c`의 집합도 바뀔까?
– `d`의 집합을 바꾸면 `c`의 집합도 바뀔까?
– `e := c` 를 수행하려면 `c`를 복사해야 하는가?
– `c` 갖고 일 봤으면, 그 집합을 없애도 되는가?

이제 C나 Java계열의 언어로 프로그래밍 하는 데에 익숙한 독자라면 머리가 복잡해 지기 시작했을 것이다. 왜냐하면, 위의 질문들에 대한 답들은, 프로그램에서 `set`과 `setUnion`을 어떻게 구현했느냐에 따라 달라지기 때문이다.

– 항상 공유하도록 구현하는 경우: 집합을 만들 때 마다 지금까지 있었던 다른 집합의 원소와 최대한 공유할 수 있도록 구현하는 것이다. 위의 프로그램에서 세번째 줄의 `setUnion(a,b)`는 `a`와 `b`가 메모리에 가지고 있는 집합 구조물을 공유하면서 합해진 집합을 만드는 것이다 (그림 3). 이렇게 되면, `a`의 집합을 바꾸면 그것을 공유하는 `c`의 집합도 바뀌게 될 것이다.
  따라서, 위의 질문에 대한 답은, `c`와 `d`의 집합은 같은 것이고, `a`나 `d`의 집합을 바꾸면 `c`의 집합이 바뀌는 여파가 생기고, `e := c`를 수행할 때 `c`의 집합이 복사되는 것이 아니고 `e`와



[그림 4] "항상 복사하기"방식의 집합 표현

같이 공유하게 된다. 마지막으로 c를 가지고 할일이 모두 끝났다고 해도 그 집합을 없앨 수 는 없다. a, b, d와 공유하고 있기 때문이다.

이렇게 항상 공유하면서 얽히고 설키도록 집합들을 구현한다면, 프로그래머는 매우 조심스러워 진다. 뭔가를 변경시키면, 그것을 공유하는 모든 것들이 변경되는 여파가 있기 때문이다. 이 방식은 메모리를 적게 소모하지만, 프로그램 작성이 매우 까다로와진다. 계산한 값들이, 프로그래머의 의도와는 다르게 다른 값으로 변경되기 쉽상이다. 생각한 대로 실행되는(버그 없는) 프로그램을 짜기는 매우 힘들어진다.

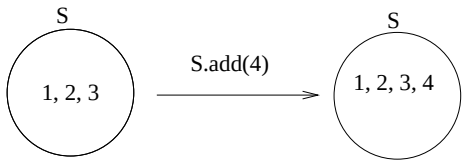
– 항상 복사하면서 구현하는 경우: 공유하면서 복잡해 지는 상황을 모면하기 위해서, 이제는 그 반대로 간 경우이다. 집합을 만들 때 마다 지금까지 있었던 다른 집합과는 전혀 공유하는 것이 없도록 한다. `set(1,2,3)`은 집합 원소 1,2,3을 가지는 구조물을 항상 새롭게 만든다. `setUnion(x,y)`는 두 집합들의 원소들을 모두 복사해서 두 집합의 합집합을 표현하는 새로운 구조물을 만든다(그림 4). 이렇게 되면 위의 질문들에대한 답을 하기는 간편해 진다. 정수값을 다루던 이전의 프로그램과 같은 경우가 되는데, c의 집합은 d의 집합과 같은 집합이고, a나 d의 집합을 바꾼다고 해도 c의 집합과는 무관하게 되고, `e := c` 를 수행할 때는 c를 복사해서 e가 가지게 된다. c 집합을 가지고 일을 마쳤으면 그 집합을 없애버려도 된다.

이런 방식으로 구현하면, 프로그램을 작성하고 이해하기는 편해진다. 프로그램이 계산하는 모든 값들은 항상 개별적으로 독립되어 있기 때문에, 값들이 얽히고 설켜있는 관계를 신경쓰면서 프로그램을 이해하거나 작성해야 하는 부담이 없다. 하지만, 문제는 메모리 소모와 시간이 많이 드는 것이다.

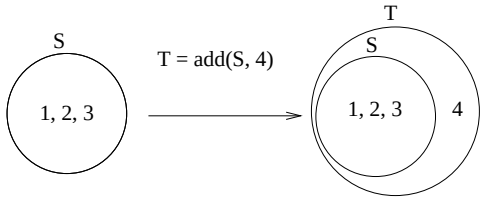
– 위의 두 방식에서 좋은 것을 취하기: 최대한 공유하면서 프로그램 작성이나 이해가 쉽도록 하는 방식이 있다. 집합을 만들 때 이미 계산된 집합들과 공유할 수 있는 것은 최대한 공유하게 하면서, 이미 계산된 집합들은 변경되지 않도록 하는 것이다. 즉, 위의 프로그램 실행중에 만드는 집합들은 다음의 세개이고, 이 세계의 집합은 앞으로 절대 변하지 않는 것이다:

$$\{1,2,3\}, \{4,5,6\}, \{1,2,3,4,5,6\}$$

이런 방안을 상정하고, 이전 질문들에 답해보자. c의 집합은 d의 집합과 같은 집합인가? 그렇다. a나 d의 집합을 바꾸는 경우는 없고, 변경된 다른 집합을 원하면 a나 d의 집합은 건드리지 않으면서, 원하는 집합을 새롭게 만든다. 이 때 지금까지 만들었던 집합들은 바뀌는 법이 없으므로, 필요하면 항상 있던 집합들을 공유할 수 잇다. `e := c` 를 수행할 때



[그림 5] 집합이라는 물건은 변한다



[그림 6] 집합이라는 값은 변하지 않는다

는 c의 집합을 e가 공유하게 한다. 마지막으로, c 집합을 가지고 일을 마쳤지만 그 집합을 없애버릴 수 는 없다. 다른 집합이 그 집합을 공유하면서 표현될 수 있기 때문이다.

위의 마지막 경우가 "값중심의 프로그래밍"인 것이다. 특히, 그렇게 구현하는 것이 자동으로 되는 프로그래밍 언어를 "값중심의 프로그래밍 언어"라고 부르자.

이야기가 필요이상으로 복잡해졌는데, 값중심(value-oriented)의 프로그래밍 스타일을 물건중심(object-oriented)의 스타일과 대비해서 요약하면 다음과 같다. 물건은 끊임없이 변하지만, 값은 일단 만들어 졌으면 변하지 않는다. 정수 1은 변하지 않는다. 1이 어느날 변해서 1.2가 되거나 99가 되지 않는다. 집합 $\{1,2\}$는 변하지 않는다. 이 집합이 변해서 $\{1,2,3\}$이 된 것이 아니고, 집합 $\{1,2\}$와 $\{1,2,3\}$은 서로 다른 집합인 것이다. 1과 1.2가 다른 값이듯이.

물건 S를 집합 $\{1,2,3\}$라고 하자. 이 집합에 원소 4를 더하는 경우

```
S.add(4)
```

물건 S는 집합 $\{1,2,3\}$ 이었다가 $\{1,2,3,4\}$ 라는 새로운 집합으로 변하게 되는 것이다(그림 5). 물건은 어떤 작업을 통해서 항상 그 상태가 변하간다.

반면에, 값중심의 프로그래밍인 경우, 집합 $S(\{1,2,3\})$라는 값에 원소 4를 추가하는 경우

```
add(S, 4)
```

새롭게 만들어 지는 집합은 S라는 집합을 변화시키지 않는다. 오직 다른 집합 $\{1,2,3,4\}$를 의미하는 것 뿐이다. 이때, 값중심의 프로그래밍에서는 값이 변하지 않으므로, 지금까지 계산된 집합들을 최대한 공유하면서 새로운 집합을 표현할 수 있을 것이다 (그림 6).

3 값중심의 프로그래밍은 특별한 것이 아니다

"값중심의 프로그래밍" 이라는 호칭이 또다른 개념의 프로그래밍 파라다임을 뜻하나 보다, 라고 긴장할 필요는 없다. 그것은 우리 프로그래머들이 까마득히 잊고 있었던, 예전(중고등학교 시절 컴퓨터 프로그래밍을 배우기 전)에는 너무도 익숙했던 프로그래밍 스타일이었다. 왜 잊고 있었던 것일까? 지금까지 익혀 온 컴퓨터 프로그래밍은 그것과 달리 복잡했기 때문이다. 왜 다르고 복잡했던 것이었을까? 그것은 자연스러운 "값중심의 프로그래밍"

을 효과적으로 지원하는 기술이 아직 컴퓨터에 없었기 때문이다. 그래서 컴퓨터 프로그래밍은 복잡한 스타일의 생각을 강요해왔던 것이다. 하지만 이제는, "값중심의 프로그래밍" 기술을 가능하게 하는 튼튼한 연구성과들이 꾸준히 쌓여서, 그러한 자연스러운 프로그래밍을 회복할 수 있게 되었다. 값중심의 프로그래밍을 제대로 지원하는 프로그래밍 언어는 이미 학교의 연구실에서 나와서, 산업 현장의 중요한 구석에서 맹활약을 해오고 있다. 이 글에서 소개할 nML이라는 언어는 그런 류의 언어들에 대해서 우리가 내놓을 수 있는 것이 되겠다. ML이라는 언어계열의 한국 사투리쯤으로 생각해도 좋다.

다시 이 섹션의 본래 논지로 돌아와서, 사실, 지금까지 300년 이상 동안 수학이라는 언어가 사용한 서술 방식이 바로 값중심의 프로그래밍이었다. 수학에서 사용하는 모든 연산은 값을 만들 뿐이지 만든 값을 바꾸지 않는다. 새로운 값을 계산하는 것 뿐이다. 어느 수학책의 한 페이지에서 따온 다음의 단락을 보자:

> "V를 $(interior\, U) \cup f^{-1}(W)$ 라고 하자.
> 그러면 $V \cap (interior\, W) = f(U)$ 가 사실임을 알 수 있다."

여기서 첫 문장에 나타나는 $interior\, U$가 U가 가지는 값을 변화시키는가? 그래서 두번째 문장에 나타나는 U는 처음의 U와 다른것이던가? 그렇지 않다. U가 가지는 값은 변하지 않는다. $interior\, U$는 U를 가지고 정의되는 어떤 새로운 값을 지칭할 뿐이지 U를 건들지는 않는 것이다. $f^{-1}(W)$라는 연산도 마찬가지다. W가 그 연산에 의해서 값이 변하는 것이 아니다. 첫문장에서나 두번째 문장에서나 W는 같은 값을 가질 뿐이다.

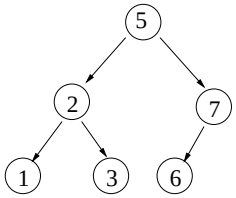
이러한 값중심의 서술방식이 수학(그리고 모든 과학) 분야에서 사람들끼리 소통하는 기본적인 프로그래밍 언어로 유구하게 이용된 이유는 뭘까? 그 이유는, 간단하고 편리해서이다. 간단하므로 편리한 것인데, 이것은 수학이나 과학이 성공한 중요한 인프라이다. 수학의 프로그램(수학의 논증들)은 그것이 옳고 그른지를 확인할 수 있을 때에만 생명이 있다. 옳고 그른지를 확인하기 편리하려면 서술하는 언어가 간단해야만 한다. 그렇게 간단하게 하는 데 기여한 언어의 주요 성질이 바로 값중심으로 프로그램하기인 것이다.

그렇다면, 컴퓨터 프로그램을 짜는 우리가 수학자들을 따라가야 할 이유는 무엇인가? 컴퓨터 소프트웨어는 수학의 프로그램에서와 똑같이, 그 참/거짓을 판명해야 하는 필요성이 명백해지고 있기 때문이다. 수학에서 프로그램의 참/거짓을 판명하는 것이 수학의 존재와 발전의 근간이었듯이, 컴퓨터 소프트웨어의 존재와 발전의 근간은 이제 프로그램의 참/거짓을 판명하는 기술이다. 프로그램이 옳고 그른지를 판명하는 것은 다름아니라 프로그램이 생각대로 작동할 지 안할 지 를 확인하는 것이다. 프로그램이 생각대로 작동한다는 것은, 프로그램이 버그없이 실행된다는 것을 뜻한다. 작성한 프로그램이 버그없이 실행될 지를 미리 자동으로 확인하는 기술, 이 기술을 달성하는 그룹이 앞으로 소프트웨어 발전의 헤게모니를 차지하게 될 텐데, 이 기술이 꽃피는 땅은 값중심의 프로그래밍 언어로 짜여진 프로그램들이 될 것이다. 수학의 참/거짓을 효과적으로 소통시켰던 언어가 값중심의 언어였다는 사실이 이 주장을 떠받치고 있다.

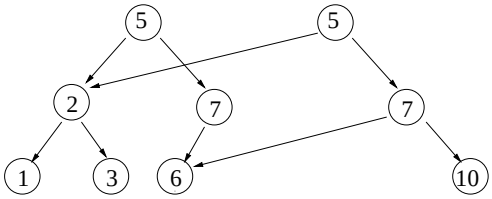
프로그램이 버그없이 실행될 지를 미리 자동으로 확인해 주는 기술이 왜 중요하고, 지금까지의 연구성과들이 어디까지 와 있고, 값중심의 프로그래밍 언어, 특히 이 글에 소개할 nML이라는 언어가 이 맥락에서 무슨 역할을 하는지에 대한 자세한 내용은 잠시뒤로 미루기로 하고, 다시 우리의 본론으로 돌아가자.

4 값중심의 프로그래밍은 비싸지 않다

값중심의 프로그래밍을 지원하려는 데 걱정되는 비용이 혹시 있지 않을까? 값중심 프로그래밍의 핵심은, 만들어진 값은 변하지 않는다, 이므로, 새로운 값을 만들때 새로운 메모리



[그림 7] 이진 검색 트리로 표현된 집합 $\{1, 2, 3, 5, 6, 7\}$



[그림 8] 이진 탐색 트리에 원소 넣기: 변하지 않는 경우

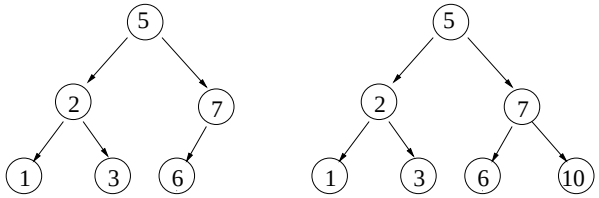
를 소모해야 하는 게 아닐까? 이 추측이 맞지 않은 이유를 구체적으로 살펴보면서 이번 첫 회를 마무리 하자.

값중심의 프로그래밍이 계산자원의 낭비를 오히려 막을 수 있는 이유는, 값중심의 프로그래밍의 핵심에 있다. 만들어진 값은 변하지 않는다, 는 핵심 덕택에, 항상 안심하고 이미 있는 값들을 공유할 수 있게된다; "철수야, 9번 버스로 통학하는구나. 너 통학하면서 9번이 다른 노선을 운행하도록 바꾸지 않을거지? 나도 않바꿔. 그래, 같이 9번 버스로 통학할 수 있겠구나." 값을 만드는 시간은 어떤가? 공유하므로 값을 반복해서 다시 만드는 시간이 줄어든다.

구체적인 프로그램으로 이야기해 보자. 예로 들었던, 정수들의 집합을 계산하는 프로그램을 생각해 보자. 정수의 집합을 구현하는 방법으로 이진 탐색 트리(binary search tree)를 이용한다고 하자. 이 방법은 집합의 원소들을 두갈래로 갈라지는(binary) 가지구조(tree)위에 특별한 방식으로 분포시켜서 원소를 찾기가(search) 빨라지도록 하는 것이다. 특별한 방식의 분포란, 갈라지는 지점(node)에 있는 원소는 항상 그 왼편에 매달린 모든 원소들보다 크거나 같고 오른편에 매달린 원소들 보다 작다. 예를들어, 집합 $\{1, 2, 3, 5, 6, 7\}$은 [그림 7]과 같이 표현되는 것이다. 그러한 조건 덕택에 어떤 원소를 찾는데 데 필요한 시간은 원소들의 총 갯수 N이 아니라 $\log N$(트리의 꼭대기에서 아래로만 내려가는 한 개 경로의 길이) 만큼만 필요하게 된다. 새로운 원소를 넣을 때에도, 넣을 위치를 찾아가야 하므로 $\log N$의 시간이 필요하다.

자 이제 따져보도록 하자. 새로운 원소를 추가해서 새로운 집합을 만들어 내는 경우를 따져보자. 우선 프로그램에서 두가지 경우가 있을 수 있겠다. 새로운 집합을 만들면 예전 집합과 새로운 집합들을 모두 유지해야 하는 경우와, 항상 새롭게 만든 집합만을 사용하고 예전의 것은 버려도 되는 경우.

– 프로그램에서 예전의 집합과 새로 만들어진 집합을 모두 유지해야 하는 경우: 값중심의 프로그래밍에서는 만들어진 집합이 변경되는 경우는 없으므로, 항상 공유할 수 있는 것은 공유하게 된다.
예를들어 위의 $\{1, 2, 3, 5, 6, 7\}$에 10이 첨가된 집합을 만드는 것을 생각하면, 새로운 집합은 3과 7번 노드에 해당하는 것만 새로 만들고 나머지는 모두 공유하면서 새로운 원소 10을 새로 만든 원소 7의 노드의 오른쪽에 넣어주면 된다 (그림 8). 따라서 시간과 공간



[그림 9] 이진 탐색 트리에 원소 넣기: 변할 수 있는 경우

이 $\log N$만큼 소모된다.

한편, 만들어진 집합이 변경될 수 있는 경우(값중심의 프로그래밍 조건을 갖추지 못한경우) 공유해서는 않된다. 그리고, 예전 집합과 새로만든 집합을 모두 가지고 있어야 하므로, 예전 집합을 그대로 복사하고 새로운 것을 하나 첨가해야 한다 (그림 9). 따라서 값중심이 아닌 경우 시간과 공간이 N만큼 소모된다. ‘

– 프로그램에서 예전의 집합은 더이상 사용하지 않고 항상 최신의 집합만 사용하는 경우: 이경우는 값중심의 프로그래밍 방식이나 그렇지 않은 방식이나 똑같은 비용이 든다. 이전 것은 더 이상 사용되지 않으므로, 현재의 집합위에 덧붙여서(destructive update) 새로운 원소를 매달면 그만이다. 드는 시간은, 새로운 원소의 위치를 찾는 시간 $\log N$(트리위의 한 경로) 만큼이다. 메모리는 새로 매다는 원소 하나만 더 있으면 된다.

구체적으로 따져보았듯이, 이진 탐색 트리로 구현된 집합에 새로운 원소를 첨가하는 연산을 구현 할 때, 값중심의 프로그래밍(값은 변하지 않는다)를 가정하면 시간과 메모리의 소모가 그렇지 않은 경우와 같거나 그보다 오히려 적게된다.

5 값중심의 프로그래밍: 정리

값중심의 프로그래밍(value-oriented programming)은 기존의 물건중심의 프로그래밍(object-oriented programming)과는 다른 생각의 틀을 제공한다. 물건(object)을 만들고 변화시키는 과정을 프로그램으로 작성하는 것이 아니고, 값(value)을 정의하고 계산하는 과정을 프로그램으로 꾸미도록 한다. 물건과 값의 차이는, 물건은 변하지만, 값은 변하지 않는다는 것이다. 항상 변화하는 물건을 생각하면서 프로그래밍 하는 복잡함 대신에, 프로그램은 값을 계산하고 정의하는 것 뿐이라는, 우리가 사실 중고등학교 수학에서 익숙하게 사용하던 쉽고 간단한 프로그래밍 스타일인 것이다. 사실 이 자연스러운 개념은 지난 300년 이상 수학(과학)의 발전을 소통시켰던 간편한 언어 인프라였고, 컴퓨터 소프트웨어의 발전을 위해서도 이러한 개념을 갖춘 언어가 사용될 필요가 있다.

우리가 이미 익숙한 바 있는 이러한 프로그래밍 언어의 개념이 지금까지 묻혀졌던 것은, 컴퓨터에서 그 개념을 효과적으로 지원하는 방법이 없었기 때문이다. 하지만, 이제는 그러한 자연스럽고 간단한 프로그래밍 개념을 효과적으로 지원하는 언어들이 이미 현장으로 나오기 시작했다.

값중심의 프로그램들은 실행 비용이 많이 드는 건 아닐까 염려할 필요는 없다. 전통적인 컴파일러 기술들이 에셈블리 프로그래밍을 만족스럽게 대체시켰듯이, 값중심의 프로그래밍 언어를 구현하는 새로운 컴파일러 기술들은 이미 그러한 염려를 기우에 가깝게 만들어 놓았다. 문제는 기존의 프로그래밍 방식의 사회/경제적 관성을 어떻게 거스르냐는 것이다.

6 nML로 경험할 수 있는 프로그래밍 기술

이제 nML이라는 값중심의 프로그래밍 언어가 컴퓨터 소프트웨어의 발달과정에서 어느 위치에서 어떤 역할을 하도록 고안되고 있는지를 살펴보도록 하겠다.

nML 프로그래밍 시스템은 소프트웨어 기술의 발달과정에서 제 2세대 기술을 충실히 갖춘 언어이다. 소프트웨어 기술의 최종 목표가 무엇이고, 이 맥락에서 겨우 "제 2세대"라는 말이 무슨 얘기인지 살펴보도록 하자.

nML로 경험할 수 있는 프로그래밍 시스템 기술중에서 C나 Java로는 겪어 볼 수 없는 것은, 프로그램의 오류를 자동으로 찾아내는 기술의 현재 수준이다. 소프트웨어의 오류를 자동으로 찾아주는 기술은 지금까지 2세대 기술이 완성되었는데, C와 Java는 1세대 기술만을 갖추고 있을 뿐이다. 1세대 오류 검증 기술은 1970년대에 달성된 것으로, 생긴게 잘못된 프로그램을 자동으로 찾아내는 기술이다. 이 기술이 문법검증기술(parsing)이고, 완전히 완성되어 어느 프로그래밍 언어에서나 제공되는 기술이다. 덕택에, 현재 누구도 문법 오류를 손으로 찾아내는 경우는 없다. C나 Java가 놓치고 있는 2세대 벅잡는 기술은 1990년대에 완성되기 시작한 기술인데, 타입검증(type checking)이라는 기술이다. 이 기술은 프로그램이 실행중에 잘못된 값이 잘못된 계산과정에 휩쓸릴 수 있는 경우가 없는지를 컴파일러가 미리 안전하게 확인해 준다. C나 Java만을 사용하는 프로그래머들은 이렇게 미리 엄밀하게 확인해 주는 실용적인 기술이 가능하다는 사실을 알지 못한다. C나 Java만을 고집하는 프로그래머들은 불안해 해야 한다. 2세대 벅잡는 기술을 맛본 프로그래머는 그러한 기술의 가능성 마져도 알지 못하고 있는 프로그래머를 쉽게 능가할 수 있는 것은 뻔하기 때문이다. 사실 이제는 3세대 벅잡는 기술 – 생긴모습도 멀쩡하고, 실행중에 잘못된 값이 흘러들지도 않지만, 실행중에 가져야할 정교한 조건을 만족시킬 수 없는 프로그램을 컴파일러가 검증해 내는 기술 – 이 실험실 수준에서 성공을 거두고 있기까지 하다. 이렇게 꾸준히 발전해가는 기술의 핵심을 경험해 본 프로그래머들의 인구가 많아질수록 우리의 프로그래밍 기술수준이 글로벌 선두를 차지할 토양은 비옥해 질 것이다.

7 2세대 벅 잡는 기술은 아직 널리 퍼지지 못했다

아쉽게도 지금 가장 널리 쓰이는 시스템 프로그래밍 언어인 C와 Java등의 언어는 2세대 벅잡는 기술을 제대로 갖추지 못한 상태이다. "재대로 갖추지 못했다"는 것은 어느정도는 컴파일러가 검증해 주지만, 많은 경우 문제가 있는 데도 불구하고 컴파일러는 아무렇지도 않게 프로그램을 통과시킨다는 것이다. 실행되면서 문제가 발생하는 것을 방치하는 것이다. 다음의 C 프로그램의 예를 보자. 메모리 포인터(x)와 그 메모리 블락의 크기(size)를 받아서 그 블락을 정수 0으로 모두 초기화하는 함수이다:

```
int init(int *x, int size)
   int i;
   for (i=0;i<size;i++)
   *(x+i) = 0;
```

이제 다음과 같이 위의 함수 init 을 사용한다고 하자:

```
init(malloc(sizeof(char)*10), 10);
```

이 코드는 타입에 맞지도 않고 위험한 프로그램이다. 메모리 블락이 정수(int)들의 블락이어야 하는데 문자(char)블락이거니와, 대게 정수를 표현하는데 4 바이트를 사용하고, 문자를 표현하는데는 1 바이트를 사용하므로, 위와같이 문자 10개를 보관할 크기의 메모리 블락을 init 함수에 던져주면 그 함수는 정수크기의 보폭으로 10발짝 움직이면서 0으로 초기화 시킨다. 할당한 메모리(10bytes) 바깥의 부분까지(40bytes) 0으로 초기화 하는 일이 벌

어지게 되는 것이다. 이러한 위험한 C 프로그램에 대해서 컴파일러는 타입검증을 제대로 못하고 경고만 줄 뿐 이다:

```
% gcc t.c
t.c: In function ‘alloc’:
t.c:11: warning: passing arg 1 of ‘init’ makes pointer from integer
without a cast
```

심지어는 프로그래머가 적절히 타입을 우기면(type cast) 위의 경고마저 불가능하게 할 수 있다:

```
init((int *)malloc(sizeof(char)*10), 10);
```

2세대 벅잡는 기술을 제대로 갖춘 언어라면 다음과 같아야 한다. 컴파일러가 주어진 프로그램을 검증해서 문제가 없다고 했다면, 정말로 실행중에 문제가 없다는 것이 보장되야 한다. 이미 이 기술을 효과적으로 구현한 실용적인 언어들이 속속 출현하고 있고, 그러한 부류의 언어들이 바로 ML 이나 Haskell 계열의 언어이며, 이 곳에 소개할 nML이 그 한 예가 된다.

이 2세대 벅잡는 기술은 아직은 문법검증(parsing)기술 만큼 모든 프로그래머들이 늘상 사용하는 기술로 널리 퍼지지는 않았으나 그러한 분별있는 프로그래밍의 세계는 곧 펼쳐질 것으로 보인다. (이러한 언어들이 실용적으로 쓰이는 예들에 대한 궁금증은 다음의 웹페이지에서 부터 풀어볼 수 있을 것이다: `http://ropas.snu.ac.kr/~kwang/functional-anger.html`) 2세대 벅잡는 기술이 프로그램 짜기를 얼마나 쉽게 해주는 지를 경험하려면 그러한 기술을 갖춘 언어로 프로그램을 해보는 수 밖에는 없다. 이와 관련해서 비록 작은 샘플이기는 하지만, 2세대 벅잡는 기술을 갖춘 nML 로 프로그램하면서 겪은 학생들의 이야기들이 `http://ropas.snu.ac.kr/~kwang/320/03/essay/` 에 모아져 있다.

사실은 이 2세대 벅잡는 기술로도 아직은 미흡한 실정이다. 실행중에 모든 값이 분별있게 착착 흘러드는 프로그램이라고 해도, 생각대로 작동하지 않을 수 있기때문이다. 타입에 맞는다는 것은 실은 초보적인 조건일 뿐이다. 휘발유만이 비행기의 엔진에 흘러든다는 것이 검증되었다고 해도, 휘발유의 폭발력이 수준미달인 경우라면 비행기가 떨어질 수 있다. 라면을 끓이는 계산에 항상 라면과 물과 불이 들어선다고 해도, 물이 한방울 뿐이거나 들어선 불이 포항제철의 용광로정도라면 라면의 맛이 망쳐질 수 있다. 타입에 맞게 컴퓨터 프로그램이 실행되더라도 생각한 것과는 다르게 진행되는 경우, 이러한 버그를 자동으로 검증하는 기술이 필요하게된다.

제 3세대 벅잡는 기술은, 이렇게 확장된 버그를 검증하는 기술을 목표로한다. 생긴모습도 멀쩡하고, 실행중에 잘못된 값이 흘러들지도 않지만, 실행중에 가져야할 정교한 조건을 만족시킬 수 없는 프로그램, 이것을 집어내는 기술이다. 이 기술의 얼개는 다음과 같다. 프로그램이 실행중에 만족해야 하는 정밀한 속사정이 엄밀한 논리식으로 정의된다. 주어진 프로그램이 실행중에 그 논리식을 거짓으로 만들 수 있는 가능성이 조금이라도 있는지를 검증한다. 있으면, 그 프로그램은 벅이 있을 수 있는 것이고, 그 가능성이 전혀 없다고 판정되면 그 프로그램은 벅이 전혀 없는 것이다. “가능성이 조금이라도 있는지”라는 표현을 쓴 것은, 벅이 없는데도 불구하고 벅이 있다고 결론내리는 경우가 생기기 때문이다. 보수적인 것이다. 하지만 안전한 것은 보장된다. 벅이 없다고 결론내려지면, 정말로 없다는 것은 보장된다. 안전하기는 하지만 완전하지는 못한 검증이다. 하지만 우리는 완전하지 못하다는 것에 만족해야 할 것 같다. 완전한 검증이 불가능하다는 것은 증명된 사실이기 때문이다.

다시 위의 C 함수 init 의 예를 보자:

```
int init(int *x, int size)
   int i;
   for (i=0;i<size;i++)
   *(x+i) = 0;
```

이 함수가 제대로 작동하기 위한 조건은 단순히 타입이 맞아야 하는 것 뿐 아니라, x가 가르키는 메모리 블락의 크기가 최소한 "정수크기×size"와 같아야 한다:

$$\text{block-size}(\texttt{x}) \geq \text{sizeof(int)} \times \texttt{size}$$

정수크기의 보폭으로 size만큼 움직이며 0으로 초기화 시키므로, 받아 논 메모리 블락은 이러한 걸음거리를 모두 포함할 정도로 넓어야 한다. 이 조건을 확인하기위해선, 위의 함수가 호출되는 곳 마다, x 와 size 가 위 조건을 만족하는 지 확인해야한다. 만일에, x 가 가르키는 블락의 크기를 미리 아는 데 실패한다면, 위의 조건을 확인할 방법이 없으므로 벅이 있는 프로그램으로 간주해야 안전하다.

프로그램이 보증해야하는 성질을 검증할 때, "잘 모르겠는" 경우를 최소화 시키고, 그 검증이 항상 자동으로, 그리고 적은 비용으로 가능하도록 하는 기술, 이러한 3세대 벅잡는 기술이 성숙되어지는 시기는 낙관적으로 잡아서 앞으로 20년정도 이후가 아닐까 생각된다. 그때가 되면 많은 사람들이 쓰는 프로그래밍 언어들은 적어도 2세대 벅잡는 기술은 제대로 갖추어진 것들이 될 것이다. 지금 우리가 1세대 벅잡는 기술을 갖춘 언어를 당연한 것으로 생각하고 쓰고 있듯이.

8 nML의 기타 특징

이러한 기술 발전의 맥락에서, nML 프로그래밍 언어는 2세대 벅잡는 기술을 제대로 갖추고 있는 상태이고, 3세대 벅잡는 기술에 대한 연구결과를 실용적으로 담아낼 대상으로 준비해 놓은 상태라고 할 수 있다. 이런 언어로 프로그램을 작성하면서 얻을 수 있는 기타 장점을 정리해 보면 다음과 같다.

- 작고 간단하다: 문법이 작고 간단할 뿐만 아니라 그 의미구조(semantics) 또한 간결하다. 지난호에 소개한 "값중심의 언어"가 가지게 되는 성질일 것이다. 극명한 예로, ML의 참고서들이 200페이지에서 최대 400페이지를 넘지 않는 반면에 Java나 C++ 의 경우는 대부분 600페이지가 넘는다.
- 안전하다: 수행중인 프로그램의 완전한 안정성이 보장되어 있다. 즉, 컴파일러를 통과한 프로그램은 실행중에 파행적으로 중단되는 경우가 (core dump 나 segmentation fault등이) 발생하지 않는다. 이것은 언어 갖추고있는 강력한 타입 시스템 때문인데, ML(Haskell도 마찬가지)에서는 특히, 타입 검사를 타입 유추를 이용해서 자동으로 하기 때문에 프로그래머가 변수의 타입을 표기할 필요가 없다. 이 기술이 위에서 얘기한 "2세대 벅잡는 기술"이다.
- 타입에 얽매이지 않는 함수도 정의할 수 있다: 하나의 함수로, 타입은 다르지만 하는 일이 같은 함수들을 표현할 수 있기 때문에 프로그램하기가 경제적이고, 타입검사를 하는 언어가 가지는 경직성이 없다. 이것도 마찬가지로 "2세대 벅잡는 기술" 덕택이다. 타입검증을 하는 Pascal이나 Java가 놓치는 기술이 되겠다.
- 편향된 프로그래밍 관성에서 벗어날 수 있다: 함수가 특이한 대상이 아니고, 여타 다른값과 구분 없이 (higher-order functions) 다루어 진다. C나 Java등의 언어에서 어느새 우리에게 스며든 프로그래밍 관성은, 함수는 특이해서 정수나 포인터등과 달리, 함수가 함수를 만들어 내거나 함수가 자유롭게 데이타로 취급되지 못한다. 함수를 저장하고 싶으면 함수 포인터를 써야하고, 포인터로 저장된 함수를 부르려면 특이하게 해야 한다.
- 명령형 프로그램도 가능하다: 메모리주소나 지정문 (assignment) 등으로 발생하는 메모리 반응들도 (side-effect들도) 아무 문제 없이 표현가능 하다.

– 정제된 모듈 처리기능을 갖추고 있다: 대형의 소프트웨어 개발에 꼭 필요한, 안전한 분리 컴파일 기능(separate compilation)이 제공되어 있다. 모듈들을 따로 컴파일해도 컴파일러가 타입오류를 안전하게 검증해 준다. 더군다나, 모듈을 일반화 시켜서 (모듈을 인자로 하는 모듈들을) 정의할 수 있기때문에, 하나의 모듈을 필요에 따라 구체화해서 재 사용할 수 있다.
  또한, 대형 프로그램 개발의 필수 덕목인, 따로따로 포장하기(modularity)와 속내용을 신경쓰지않게하기(data abstraction) 등을 프로그래머가 제대로 잘 적용하고 있는지를 컴파일러가 검증해준다. 프로그래머가 모듈을 정의할 때 위의 덕목에 맞도록 정의하게 되는데, 그 모듈을 사용하는 파트에서 덕목에 어긋나게 사용되는 경우가 혹시 있는지 자동으로 검증해 준다. 이 검증도 nML이 가지고 있는 타입 시스템 덕택에 가능하게 된다.
– 자동으로 메모리를 관리해 준다: 효율적인 메모리 재활용 시스템이 (garbage collection이) 제공되기 때문에 프로그래머가 메모리의 부족이나 재사용등에 대한 부담에서 자유롭다.
– 예외상황 관리(exception handling)이 간단하다: 이것을 이용 해서, 프로그램의 실행 중에 발생할 수 있는 긴급상황들을 (예를 들면, x/y 에서 y 값이 0인 경우들을) 정황에 맞게 프로그래머가 적절히 발생시키고 처리할 수 있다. 특히 nML에서는 발생하는 데 처리되지 못할 수 있는 예외상황이 어떤 것이 있는지 컴파일러가 자동으로 검증해 준다. Java에서와 같이 프로그래머가 함수마다 어떤 예외상황이 발생할 수 있는지 선언할 필요가 없다.
– 정형적인 의미구조를 갖추고 있다: 프로그램을 이해하는 데 필요한 정확한 정의가 제공되어 있다. 따라서, 프로그램의 의미에 혼동의 여지가 없으므로 프로그램을 분석하고 관리하는 작업이 용이하다.
– 프로그래밍의 미학을 익히기 쉽다: 대형의 고난도 프로그램이 드러내는 복잡성을 다룰 수 있는 프로그래밍 원리와 미학이 자연스럽게 표현된다. "값중심의 프로그래밍"을 지원하기 때문에 주장할 수 있는 덕목일 것이다.

이것으로 nML에 대한 간단한 소개를 마치기로하자. 짧게 정리해 보면, nML로 프로그램을 짜면, 값중심의 간단한 생각으로 프로그램을 짜면서 실용적인 프로그램을 구축할 수 있을 뿐 아니라, 현재의 프로그래밍 기술수준(2세대 벅잡는 기술)의 제대로된 모습을 경험하게 될 것이라는 것이다.

필자는 지난 4년간 nML 프로그래밍 시스템을 이용해서 서울대와 KAIST 학부 3학년 대상의 프로그래밍 언어 강의를 운영해 왔는데, 여기서 학생들의 소감을 있는그대로 전하면서 이 글을 마친다.

9 nML을 실습언어로 사용한 프로그래밍 언어 수강생들의 소감

– "여타언어로는 구현하기 힘든 것들이 nML로는 쉽게 구현되는 것을 보고 참 신기했던 기억이 납니다"
– "만약 어떤한 컴퓨터 랭귀지도 모르는 상태에서 nML이 자연스럽게 받아질 수 있을까 묻는 다면 Yes로 대답할 것이다. 다른 언어보다 쉽고 뛰어나다는 것은 아니다."
– "다른 과목 숙제, 프로젝트에서 C나 Java로 짜여 되는 걸 보면, '아 nML로 짜고 싶다'는 생각이 듭니다. nML프로그래밍이 지금은 확실히 편하고 생각하기도 편하구요."
– "nML 프로그래밍은 생각하는 방식이 너무 재미있습니다."
– "nML은 굉장히 인간중심, 개념 중심의 언어라고 느꼈습니다."
– "초기에는 당황하고 어색한 점이 많았습니다. 하지면 하면 할수록, 내가 생각하는 바를 그대로 프로그램으로 옮길 수 있다는 큰 장점을 발견할 수 있었습니다."

– “C나 C++등을 짜면서 생각이 점점 복잡하게 꼬여서 프로그램 짜는 데 애먹는 경우가 많았는데 nML은 간단하면서도 원하는 결과를 정확하게 얻어내서 프로그래밍 하기가 좋았습니다.”
– “nML 프로그래밍 하면서 느꼈던 점 중에 가장 인상 깊었던 것은 일단 컴파일을 성공하고, 프로그램이 돌아갈 것이라 기대하고 샐행했을 때 정말 ’제대로’ 돌아가는 경우가 많았다는 점입니다. C++등의 언어는 컴파일 에러를 모두 없앤 후에, 런타임 에러를 없애는 데에도 많은 시간이 소비되지만, nML 프로그래밍에서는 일단 컴파일이 성공하면 대부분 내 생각대로 돌아가니까 처음에는 매우 신기했습니다.”
– “nML은 명령어가 많지 않아서 쉽게 알 수 있었습니다.”
– “nML을 하면서 정말 프로그램을 짜고 깔끔한 느낌이 든다는 것이 가장 인상적이네요. C++같은거에선 좀 느끼기 힘들었죠.”
– “슬슬 C로 짜는게 외 불편한지 느껴지기도 합니다.”
– “멋지다. 특히 디버깅 할때. 솔직히 디버깅 할 것도 없게 해주는 것 같지만.”
– “무엇보다도 nML이 우리의 언어라는 것이 자랑스럽습니다!”
– “nML을 처음 접했을 때 ’참 어렵다 그리고 이런 언어를 왜 쓰나’ 하는 생각을 했습니다. 하지만 시간이 조금씩 지나고, C++로 프로그램을 짜다보니 ’아 이거 nML로 쓰면 대개 편할텐데’라는 생각이 자주 들었습니다. nML이 좀 더 프로그래밍 언어에 대한 시각을 넓히는 데 도움을 주었습니다.”
– “뭔가 딱딱 맞아 들어가는 것이 지금까지와 다른 프로그래밍의 원리를 배울 수 있었습니다.”
– “퍼즐이 맞춰질 때의 그 쾌감같이 하나하나 내 생각되로 움직이는 게 재밌었습니다. nML이란 언어 참 편한거 같아요. 처음에는 익숙치 않아서 적응하기 힘들었는데 지금은 C나 Java보다 편리하다는 것이 느껴집니다. C에서는 너무 생각해야 될것이 많거든요. 어제 프로젝트 하는데 그놈의 세그폴 때문에.”
– “값중심의 프로그래밍. 처음에는 지금까지 흔히 쓰던 assignment가 되지 않아서 많이 힘들었습니다. 하지만 nML적 프로그래밍 방법이 훨씬 더 수학적 논리적으로 맞다는 사실을 알고서는, 아직은 부족하지만 nML이 정말 안정적인 언어라는 것을 알았습니다. type checking 또한 잘 되는 언어라는 점에서 상당히 안전한 언어입니다.”
– “C에 비해 월등히 사용하기 편리했고, 좋았습니다.”
– “아직도 nML프로그래밍에 그리 능숙하지는 않지만 처음 nML 프로그래밍을 했을 때에는 정말 생소해서 많이 어렵게 느껴졌습니다. 그리고 처음엔 C나 JAVA와 비교해서 이렇게 단순하게 생긴 언어로 충분히 복잡한 프로그래밍을 할 수 있을지 의문스러웠습니다. 그렇지만 지금은 오히려 C나 자바보다 더 간단하게 프로그래밍을 할 수 있을 것 같은 생각이 듭니다. conputer science를 공부하는 데 있어서 C나 JAVA보다 nML을 먼저 접한다면 더 도움이 될 것 같기도 합니다.”
– “매우 간단하고 깔끔하고 논리적인 언어였다. 프로그래밍의 아름다움을 느낄 수 있는 그런 언어였다.”
– “자료구조나 알고리즘을 그대로 표현할수 있다는게 인상깊었습니다. 다른언어를 써온 사람이라면 처음에는 기존의 언어에 익숙해져 있어서 nML 등의 언어의 장점, 유용함 등을 단번에 알기는 힘들것 같지만 익숙해지고 누구나 편리하다고 느낄만한 언어인것 같아요. 다른언어에 비해 매우 짧은 코드도 인상적이었습니다.”
– “nML은 불과 한달사이에 나를 마약처럼 사로잡아 버렸다. 메모리도 수많은 변수들도 신경쓸 필요가 없었다. 프로그래밍 내내 돌아다니는 것은 논리 그 이상도 이하도 아니었다. 생각하는 것을 바로 기호로 나타내고 컴퓨터는 방금 내가 했던 생각을 그대로 따라 원하

는 결과값을 얻어낸다. 어떤것이 좋은 언어인지는 모르지만 잘 만들어진 언어라는 말은 바로 이런경우에 쓰는 것이라는 느낌이 들었다."

– "처음에 nML로 수업을 진행한다고 했을 때 새로운 언어를 배워야 하는 부담감이 있었던 것이 사실입니다. 하지만 언어를 배워가면서 첫번째 숙제를 무난히 끝내고 나서는 오히려 nML이라는 언어가 이전의 언어보다 훨씬 재밌었고, 숙제가 기다려 졌습니다. nML이라는 언어는 이전의 언어와는 다르게 프로그래밍이 쉬우면서도 그 기능은 강력합니다. 내가 생각하는 것을 표현하고 나서, 그 생각에 문제가 없다면 사사로운 실수등을 잘 잡아주어서 프로그래밍하기가 수월했습니다."

감사의 글

이 글은 저자의 홈페이지에 걸린 글들과 [마이크로 소프트웨어] 2002년 6월호와 7월호에 이미 연재한 바 있는 내용에 기초하였다.

이광근

– 2003-현재, 부교수, 컴퓨터공학부, 서울대
– 1995-2003, 조교수/부교수, 전산학과, KAIST
– 1993-1995, Member of Technical Staff, Software Principles Research Dept., Bell Laboratories, Murray Hill, U.S.A.
– 1993, Ph.D., Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign
– 1987, B.S., Dept. of Computer Science & Statistics, Seoul National University

<관심분야> 프로그램 분석, 프로그래밍 언어