값중심의 프로그래밍
value-oriented programming*

이광근
cs.kaist.ac.kr/~kwang
프로그램분석 시스템 연구단
전산학과
KAIST

[마이크로 소프트웨어], 2002년 6월호 [nML과 함께하는 프로그래밍 여행: 제 1 편]

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

*Andrew Appel (www.cs.princeton.edu/~appel)이 1994년에 Bell Lab에서 진행한 세미나에서 처음으로 사용하였고, 그 후 "object-oriented programming"과 대비되서 널리 통용되는 용어임.

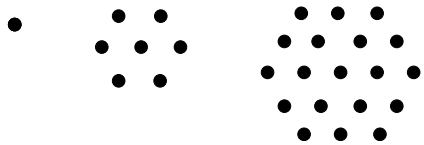

그림 1: 육각형수

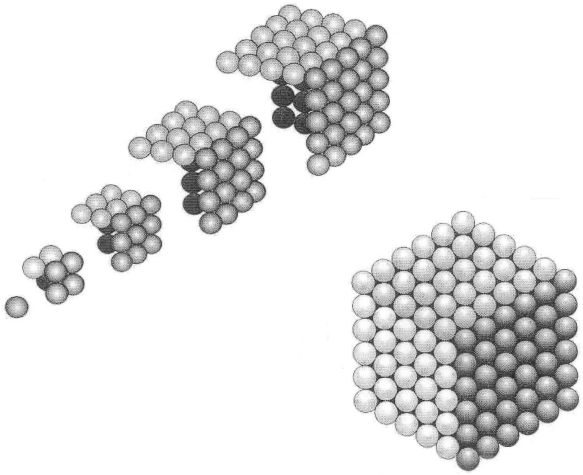

그림 2: 육각형수는 정육면체의 한꺼풀. 위와같이 3면체의 함 꺼풀씩을 쌓아가면 꽉찬 정육면체를 만들 수 있는데, 각각의 3면체를 오른쪽 중앙의 꼭지점을 중심으로 보면 바로 육각형수(오른쪽 아래) 이다. 참조: *What is Intelligence?*, Jean Khalfa (ed.), Cambridge University Press, 1994

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 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.

이번 [nML과 함께하는 프로그래밍 여행] 시리즈에서는 프로그래머의 생각의 틀을 다양하게 해 주는 프로그래밍 언어로 nML이란 것을 소개하려고한다. 이번 첫 기사에서는 nML이 안내하는 생각의 틀은 어떤 것인지를 우선 살펴보도록 한다. 제목에서 눈치챘겠지만, 그것은 값중심으로 생각하기이다(value-oriented programming).

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?
- 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 c가 바뀌지는 않는다. c는 3이라는 값을 계속 가지고 있게된다. 마찬가지로 d를 바꾼다고 해서 c가 가진 값 3이 바뀌지는 않는다. e := c를 수행하면 3이라는 c의 값이 e에 저장된다. c 갖고 일 봤으면, c가 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 x와 y를 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 c의 집합이 d의 집합과 같은가?

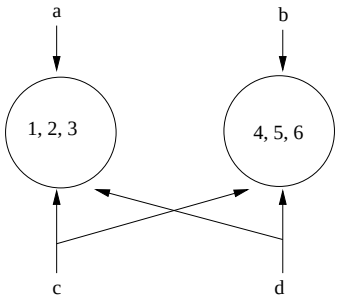

그림 3: "항상 공유하기"방식의 집합 표현

- a의 집합을 바꾸면 c의 집합도 바뀔까?
- d의 집합을 바꾸면 c의 집합도 바뀔까?
- `e := c` 를 수행하려면 c를 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 set과 setUnion을 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 a와 b가 메모리에 가지고 있는 집합 구조물을 공유하면서 합해진 집합을 만드는 것이다 (그림 3). 이렇게 되면, a의 집합을 바꾸면 그것을 공유하는 c의 집합도 바뀌게 될 것이다.

  따라서, 위의 질문에 대한 답은, c와 d의 집합은 같은 것이고, a나 d의 집합을 바꾸면 c의 집합이 바뀌는 여파가 생기고, `e := c`를 수행할 때 c의 집합이 복사되는 것이 아니고 e와 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 새로

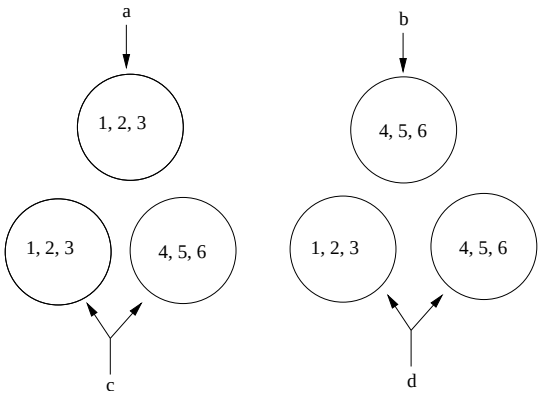

그림 4: "항상 복사하기"방식의 집합 표현

운 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 때는 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.

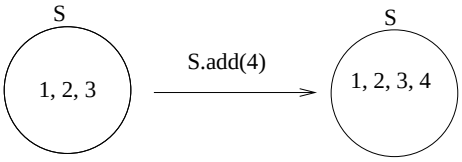

그림 5: 집합이라는 물건은 변한다

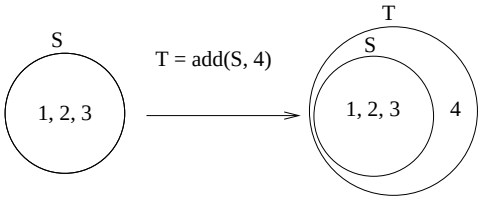

그림 6: 집합이라는 값은 변하지 않는다

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 지원

하는 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 시리즈에서 소개할 nML이라는 언어는 그런 류의 언어들에 대해서 우리가 내놓을 수 있는 대표선수이다. 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 시리즈에서 소개할 nML이라는 언어는 이 맥락에서 무슨 역할을 하

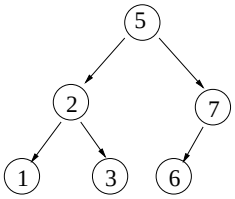

그림 7: 이진 검색 트리로 표현된 집합 $\{1, 2, 3, 5, 6, 7\}$

는지에 대한 자세한 내용은 다음호에 계획되어 있으므로 그때로 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 메모리를 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 만들어내는 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

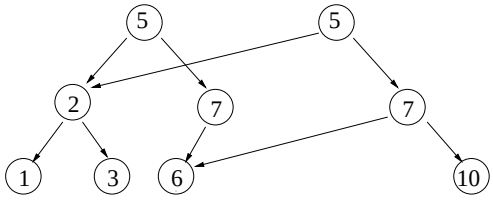

그림 8: 이진 탐색 트리에 원소 넣기: 변하지 않는 경우

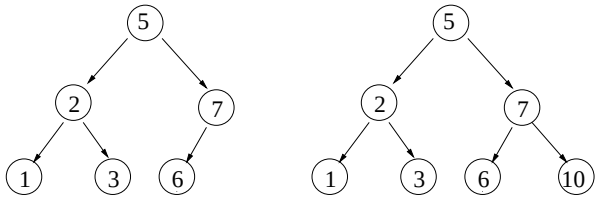

그림 9: 이진 탐색 트리에 원소 넣기: 변할 수 있는 경우

모두 공유하면서 새로운 원소 10을 새로 만든 원소 7의 노드의 오른쪽에 넣어주면 된다 (그림 8). 따라서 시간과 공간이 $\log N$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 마치면서

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 생각하면

서 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 300년이상 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. 그것을 거스르는 해커들, 이미 짜여진 매트릭스를 거슬러 새로운 프로그래밍의 세계를 확인하고 싶어하는 레오[1]들이 나설 무대는 준비되어 있다.

이쯤해서 이번 첫회의 내용을 마치도록 하자. 다음호에서는, 이 시리즈가 다루기로한 nML이라는 값중심의 프로그래밍 언어가 컴퓨터 소프트웨어의 발달과정에서 어느 위치에서 어떤 역할을 하도록 고안되고 있는지를 살펴보도록 하겠다. 그렇게 두 회의 글을 통해서 nML을 보는 우리의 안목을 준비한 후에, 3회 부터는 구체적인 nML 프로그래밍의 세계로 여행해보도록 하자.

[1] 영화 [매트릭스]를 기억하시기를.