nML과 함께 하는 프로그래밍 여행 1

다른 생각의 틀 값 중심의 nML 프로그래밍

이광근 kwang@cs.kaist.ac.kr, http://cs.kaist.ac.kr/~kwang
KAIST 전산학과 교수로 재직하고 있으며, 1998년 가을부터 과기부 창의적연구진흥과제 지정 '프로그램 분석 시스템 연구단(ropas.kaist.ac.kr)'을 맡고 있다.

이번 호부터 프로그래머의 생각의 틀을 다양하게 해주는 프로그래밍 언어로 nML을 소개하려고 한다. 첫 시간에는 nML이 안내하는 생각의 틀은 어떤 것인지를 우선 살펴보도록 하겠다. 제목에서 눈치 챘겠지만, 그것은 값 중심으로 생각하는 것이다.

편집자 주: 많은 개발자들에게 숨겨진 가치 있는 언어를 소개하고자 하는 의도로 시작된 Haskell에 이어 이번 호부터 소개되는 nML 역시 함수를 잘 지원하는 언어 중 하나다. 현재 이 언어에 대한 연구는 KAIST의 ROPAS(http://ropas.kaist.ac.kr/n/)에서 진행되고 있다. nML의 프로그램 소스와 실행 파일은 ROPAS 홈페이지에서 다운받을 수 있다.

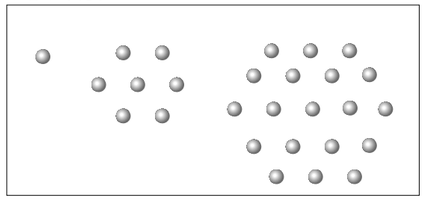
생각이 바뀌면 어려웠던 문제가 쉽게 풀린다. 문제를 바라보는 시각이 조정되는 순간에, 예전에 어렵게 보였던 문제가 쉽게 풀리는 것이다.

◆ **문제 1** : 50킬로미터 떨어져 있는 자전거 두 대가 마주보며 시속 25킬로로 달리기 시작했다. 마주달리는 두 대의 자전거 사이를 부지런히 왕복하는 파리가 있다. 파리는 시속 50킬로로 비행한다. 자전거가 마주칠 때까지 파리가 비행한 거리는?

철수의 답 : 철수는 수열과 극한, 미분과 적분을 이용한다. 따라서 앞의 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. 철수처럼 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 충돌 때까지의 시간과 같다. 그 시간은 1시간이고, 따라서 파리의 총 비행거리는 50킬로미터이다. 순이는 문제를 해결하는 데 필요한 만큼의 상위의 수준에서 상황을 정의하고 해결하는 언어를 구사하는 것이다.

〈그림 1〉 육각형수

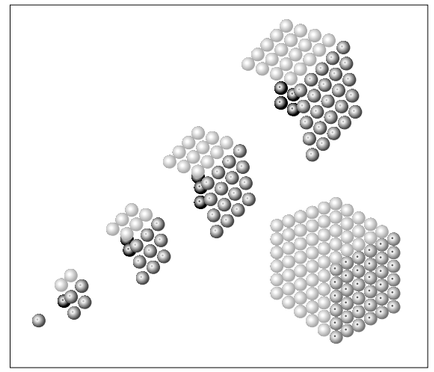


◆ **문제 2** : 육각형 정수는 1, 7, 19, 37, 61, 91, 127 등인데, 정육각형 모양이 되도록 바둑판 위에 놓아야 할 바둑알들의 개수이다(〈**그림 1**〉). 이러한 육각형 정수들을 처음부터 합해가면 항상 어느 정수의 3승이 된다고 한다. 사실인가?

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 그림이 정육면체의 한 꺼풀에 해당하고, 그 한 꺼풀들이 차곡차곡 쌓이면 항상 꽉 찬 정육면체를 만든다는 것을 보인다(〈**그림 2**〉) 따라서 육각형 정수들을 차례대로 합하면, 정확하게 어떤 정육면체를 꽉 채우는 알갱이들의 개수가(즉, 어떤 수의 3승이) 되는 것이다. 2차원과 3차원 도형이 순이가 사용한 프로그래밍 언어인 것이다.

〈그림 2〉 육각형수는 정육면체의 한 꺼풀



참조 : What is Intelligence?, Jean khalfa(ed.), Cambridge University Press, 1994

이렇듯 문제를 정의하고 답을 궁리하는 생각의 틀을 적절히 선택할 필요가 있다. 그러면 만들어지는 해답은 작고 간단해지고 이해하기 쉽기 때문이다. 그렇다면 컴퓨터 프로그래밍의 세계에서 문제를 정의하고 답을 궁리하는 생각의 틀은 무엇으로 결정될까? 다른 많은 것들이 있겠지만, 크게 영향을 미치는 것이 프로그래밍 언어일 것이다.

값 중심의 프로그래밍

다음의 프로그램을 생각해 보자.

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

프로그램이 실행됐다고 하고, 몇 가지 질문을 던져보자.

◆ c의 3이 d의 3인가?
◆ a를 바꾸면 c도 바뀔까?
◆ d를 바꾸면 c도 바뀔까?
◆ e := c를 수행하려면 c를 복사해야 하는가?
◆ c를 갖고 일 봤으면, 그 3을 없애도 되는가?

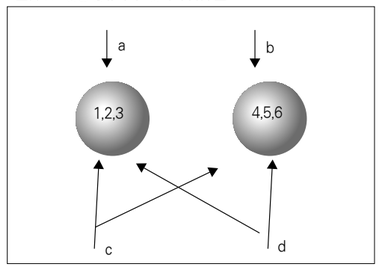
이러한 질문들은 싱겁기 그지없다. 답하기 쉬운 이유는 앞의 프로그램에서 다루는 값이 간단한 것(정수)이기 때문이다. c의 3은 d의 3과 같다. 프로그램 수행 후에 a를 바꾼다고 해서 c가 바뀌지는 않는다. c는 3이라는 값을 계속 갖고 있게 된다. 마찬가지로 d를 바꾼다고 해서 c가 가진 값 3이 바뀌지는 않는다. e := c를 수행하면 3이라는 c의 값이 e에 저장된다. c를 갖고 일 봤으면, c가 가진 값 3은 지워버려도 된다.

이제, 정수보다는 복잡한(컴퓨터에 구현하려할 때 할 일이 많은) 값을 다루는 프로그램에 대해서 비슷한 질문을 해보자. 집합을 다루는 프로그램을 생각하자. 다음 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);
```

프로그램이 실행된 후, 좀 전과 같은 질문을 해 보자.

〈그림 3〉 '항상 공유하기' 방식의 집합 표현



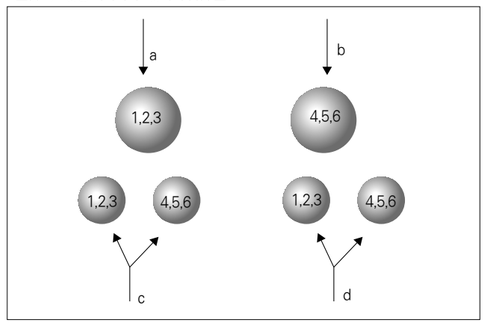
◆ 앞의 프로그램에서 c의 집합이 d의 집합과 같은가?
◆ a의 집합을 바꾸면 c의 집합도 바뀔까?
◆ d의 집합을 바꾸면 c의 집합도 바뀔까?
◆ e := c를 수행하려면 c를 복사해야 하는가?
◆ c를 갖고 일 봤으면, 그 집합을 없애도 되는가?

이제 C나 자바 계열의 언어로 프로그래밍 하는 데에 익숙한 독자라면 머리가 복잡해지기 시작했을 것이다. 왜냐하면, 앞의 질문들에 대한 답들은 프로그램에서 set과 setUnion을 어떻게 구현했느냐에 따라 달라지기 때문이다.

◆ **항상 공유하도록 구현하는 경우** : 집합을 만들 때마다 지금까지 있었던 다른 집합의 원소와 최대한 공유할 수 있도록 구현하는 것이다. 앞의 프로그램에서 세 번째 줄의 setUnion(a, b)는 a와 b가 메모리에 갖고 있는 집합 구조물을 공유하면서 합해진 집합을 만드는 것이다(〈**그림 3**〉). 이렇게 되면, a의 집합을 바꾸며 그것을 공유하는 c의 집합도 바뀌게 될 것이다. 따라서 앞의 질문에 대한 답을 말하면 c와 d의 집합은 같은 것이고, a나 d의 집합을 바꾸면 c의 집합이 바뀌는 여파가 생기고, e := c를 수행할 때 c의 집합이 복사되는 것이 아닌 e와 같이 공유된다. 마지막으로 c를 가지고 할 일이 모두 끝났다고 해도 그 집합을 없앨 수는 없다. a, b, d와 공유하고 있기 때문이다.
이렇게 항상 공유하면서 얽히고 설키도록 집합을 구현한다면, 프로그래머는 매우 조심스러워 진다. 뭔가를 변경하면, 그것을 공유하는 모든 것들이 변경되는 여파가 따르기 때문이다. 이 방식은 메모리를 적게 소모하지만, 프로그램 작성이 매우 까다로워진다. 계산한 값들이 프로그래머의 의도와는 다르게 다른 값으로 변경되기 십상이다. 생각한 대로 실행되는**(버그 없는)** 프로그램을 짜기는 매우 힘들어진다.

◆ **항상 복사하면서 구현하는 경우** : 공유하면서 복잡해지는 상황을 모면하기 위해 이제는 그 반대로 간 경우이다. 집합을 만들 때마다 지금까지 있었던 다른 집합과는 전혀 공유하는 것이 없도록 한다. set(1,2,3)은 집합 원소 1,2,3을 갖는 구조물을 항상 새롭게 만든다. setUnion(x,y)는 두 집합들의 원소들을 모두 복사해서 두 집합의 합집합을 표현하는 새로운 구조물을 만든다(〈**그림 4**〉). 이렇게 되면 앞의 질문들에 대한 답을 하기는 간편해진다. 정수 값을 다루던 이

〈그림 4〉 '항상 복사하기' 방식의 집합 표현



전의 프로그램과 같은 경우가 되는데, c의 집합은 d의 집합과 같은 집합이고, a나 d의 집합을 바꾼다고 해도 c의 집합과는 무관하게 되고, e := c를 수행할 때는 c를 복사해서 e가 갖게 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.

◆ **앞의 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, 앞의 프로그램 실행 중에 만드는 집합들은 다음의 세 개이고, 이 세 개의 집합은 앞으로 절대 변하지 않는 것이다.

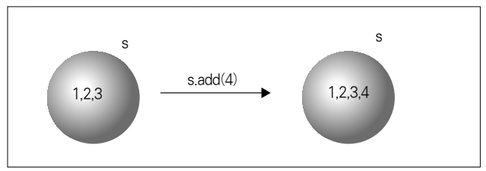
```
{1,2,3}, {4,5,6}, {1,2,3,4,5,6}
```

이런 방안을 상정하고 이전 질문들에 대해 답해 보자. 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 있다. e := c를 수행할 때는 c의 집합을 e가 공유하게 한다. 마지막으로 c 집합을 가지고 일을 마쳤지만 그 집합을 없애버릴 수는 없다. 다른 집합이 그 집합을 공유하면서 표현될 수 있기 때문이다.

앞의 마지막 경우가 '값 중심의 프로그래밍'인 것이다. 특히, 그렇게 구현하는 것이 자동으로 되는 프로그래밍 언어를 '값 중심의 프로그래밍 언어'라고 부르자.

이야기가 필요 이상으로 복잡해졌는데, 값 중심(Value-Oriented)의 프로그래밍 스타일을 물건 중심(Object-Oriented)의 스타일과 대비해서 요약하면 다음과 같다. 물건을 중심으로 생

〈그림 5〉 집합이라는 물건은 변한다.



각하는 경우를 우선 생각하자. 물건 S를 집합 {1,2,3}이라고 하자. 이 집합에 원소 4를 더하는 경우를 보자

```
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 변해간다. 그렇다면 값 중심의 프로그래밍인 경우 집합 S({1,2,3})라는 값에 원소 4를 추가해 보자

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

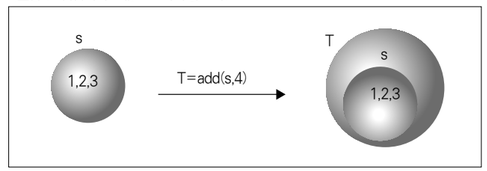
새롭게 만들어지는 집합은 S라는 집합을 변화시키지 않는다. 오직 다른 집합 {1,2,3,4}를 의미하는 것뿐이다. 이때, 값 중심의 프로그래밍에서는 값이 변하지 않으므로 지금까지 계산된 집합들을 최대한 공유하면서 새로운 집합을 표현할 수 있을 것이다(〈**그림 6**〉). 물건은 끊임없이 변하지만, 값은 일단 만들어지면 변하지 않는다. 정수 1은 변하지 않는다. 1이 어느 날 변해서 1.2가 되거나 99가 되지 않는다. 집합 {1,2}는 변하지 않는다. 이 집합이 변해서 {1,2,3}이 되지 않고, 집합 {1,2}와 {1,2,3}은 서로 다른 집합인 것이다. 1과 2가 다른 정수 값이듯이.

값 중심의 프로그래밍은 특별한 것이 아니다

'값 중심의 프로그래밍'이라는 호칭이 '또 다른 개념의 프로그래밍 패러다임을 뜻하나 보다'라고 긴장할 필요는 없다. 그것은 우리 프로그래머들이 까마득히 잊고 있었던, 이전(컴퓨터 프로그래밍을 배우기 전)에는 너무도 익숙했던 프로그래밍 스타일이었기 때문이다. 이미 친숙했던 개념에 색다른 이름을 붙이는 이유는 잊고 있었던 개념을 일깨우기 위한 구령이 필요하기 때문이다.

그럼 잊고 있었던 이유는 뭘까? 지금까지 컴퓨터 프로그래밍을 배우면서 머리에 쌓아간 개념은 그것과는 다른 복잡한 것이었기 때문이다. 왜 다른 것이었을까? 그것은 자연스러운 값 중심의 프로그래밍을 효과적으로 지원하는 기술이 컴퓨터에 없었기 때문이

〈그림 6〉 집합이라는 값은 변하지 않는다.



었다. 하지만 그러한 기술을 가능하게 하는 견고한 연구 성과들이 꾸준히 쌓여, 지금은 그러한 자연스러운 프로그래밍을 회복할 수 있게 되었다. 값 중심의 프로그래밍을 제대로 지원하는 프로그래밍 언어는 이미 학교의 연구실에서 나와 실제 일선의 프로그래밍 언어로서 중요한 곳에서 맹활약을 해오고 있다. 이 연재에서 소개할 nML이라는 언어는 그런 류의 언어들에 대해 우리가 내놓을 수 있는 대표선수이다. ML이라는 언어 계열의 한국 사투리쯤으로 생각해도 좋다.

다시 본래 논지로 돌아와서, 지금까지 300년 이상 동안 수학이라는 언어가 사용한 서술 방식이 바로 값 중심의 프로그래밍이었다. 수학에서 사용하는 모든 연산은 값을 만들 뿐이지 만든 값을 바꾸지 않는다. 새로운 값을 계산하는 것뿐이다. 어느 수학책의 한 페이지에서 따온 다음의 단락을 보자.

"V를 (interior U) U f^{-1}(W)라고 하자
그러면 V (interior W) = f(U)가 사실임을 알 수 있다 "

여기서 첫 문장에 나타나는 *interior U*는 *U*가 갖는 값을 변화시키는가? 그래서 두 번째 문장에 나타나는 *U*는 처음의 *U*와 다른 것이던가? 그렇지 않다. *U*가 가지는 값은 변하지 않는다. *interior U*는 *U*를 가지고 정의되는 어떤 새로운 값을 지칭할 뿐이지 *U*를 건들지는 않는 것이다. $f^{-1}(W)$라는 연산도 마찬가지다. *W*가 그 연산에 의해서 값이 변하는 것이 아니다. 첫 문장에서나 두 번째 문장에서나 *W*는 같은 값을 가질 뿐이다.

이러한 값 중심의 서술 방식이 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 인프라다. 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 값 중심 프로그래밍인 것이다.

그렇다면 컴퓨터 프로그램을 짜는 우리가 수학자들을 따라가야

할 이유는 무엇인가? 컴퓨터 소프트웨어는 수학의 프로그램에서와 똑같이, 그 참과 거짓을 판명해야 하는 필요성이 명백해지고 있기 때문이다. 수학에서 프로그램의 참과 거짓을 판명하는 것이 수학의 존재와 발전의 근간이었듯이, 컴퓨터 소프트웨어의 존재와 발전의 근간은 이제 프로그램의 참과 거짓을 판명하는 기술이다. 프로그램이 옳고 그른지를 판명하는 것은 다름 아니라 프로그램이 생각대로 작동할지 안 할지를 확인하는 것이다. 프로그램이 생각대로 작동한다는 것은 프로그램이 버그 없이 실행된다는 것을 뜻한다. 작성한 프로그램이 버그 없이 실행될지를 미리 자동으로 확인하는 기술, 이 기술을 달성하는 그룹이 앞으로 소프트웨어 발전의 헤게모니를 차지하게 될 텐데, 이 기술이 꽃피는 땅은 값 중심의 프로그래밍 언어로 짜여진 프로그램들이 될 것이다. 수학의 참과 거짓을 효과적으로 소통시켰던 언어가 값 중심의 언어였다는 사실이 이 주장을 견고히 떠받치고 있다.

프로그램이 버그 없이 실행될지를 미리 자동으로 확인하는 기술이 왜 중요하고, 지금까지의 연구 성과들이 어디까지 와있고, 값 중심의 프로그래밍 언어, 특히 이 연재에서 소개할 nML이라는 언어는 이 맥락에서 무슨 역할을 하는지에 대한 자세한 내용은 다음호에 계획되어 있으므로 미루기로 하고 다시 본론으로 돌아가자.

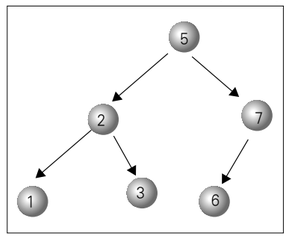
값 중심의 프로그래밍은 비싸지 않다

값 중심의 프로그래밍을 지원하려는 데 걱정되는 비용이 혹시 있지 않을까? 값 중심 프로그래밍의 핵심은 '만들어진 값은 변하지 않는다'이므로 새로운 값을 만들 때 새로운 메모리를 소모해야 하는 게 아닐까? 이 추측이 맞지 않은 이유를 구체적으로 살펴보면서 이번 호를 마무리하자.

값 중심의 프로그래밍이 계산자원의 낭비를 오히려 막을 수 있는 이유는 값 중심의 프로그래밍의 핵심에 있다. '만들어진 값은 변하지 않는다'는 핵심 덕택에 항상 안심하고 이미 있는 값들을 공유할 수 있게 된다. "철수야, 9번 버스로 통학하는구나. 너 9번 노선 바꾸지 않을 거지? 나도 안 바꿔. 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 원소들

〈그림 7〉 이진 검색 트리로 표현된 집합 {1,2,3,5,6,7}



보다 크거나 같고 오른편에 매달린 원소들보다 작다. 예를 들어, 집합 {1,2,3,5,6,7}은 **〈그림 7〉**과 같이 표현되는 것이다.

그러한 조건 덕택에 새로운 원소를 찾는 데에 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 N의 시간이 필요하다.

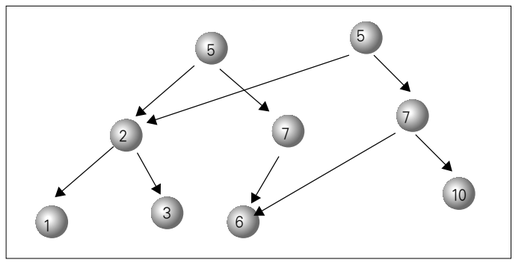
자, 이제 따져보도록 하자. 새로운 원소를 추가해서 새로운 집합을 만들어 내는 경우를 따져보자. 우선 프로그램에서 두가지 경우가 있을 수 있겠다. 새로운 집합을 만들면 예전 집합과 새로운 집합들을 모두 유지해야 하는 경우와, 항상 새롭게 만든 집합만을 사용하고 예전의 것은 버려도 되는 경우이다.

◆ **프로그램에서 예전의 집합과 새로 만들어진 집합을 모두 유지해야 하는 경우** : 값 중심의 프로그래밍에서는 만들어진 집합이 변경되는 경우는 없으므로, 항상 공유할 수 있는 것은 공유하게 된다. 예를 들어 앞의 {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 공간이 log N만큼 소모된다. 한편, 만들어진 집합이 변경될 수 있는 경우(값 중심의 프로그래밍 조건을 갖추지 못한 경우) 공유해서는 안된다. 그리고 예전 집합과 새로 만든 집합을 모두 가지고 있어야 하므로, 예전 집합을 그대로 복사하고 새로운 것을 하나 첨가해야 한다(**〈그림 9〉**). 따라서 값 중심이 아닌 경우 시간과 공간이 N만큼 소모된다.

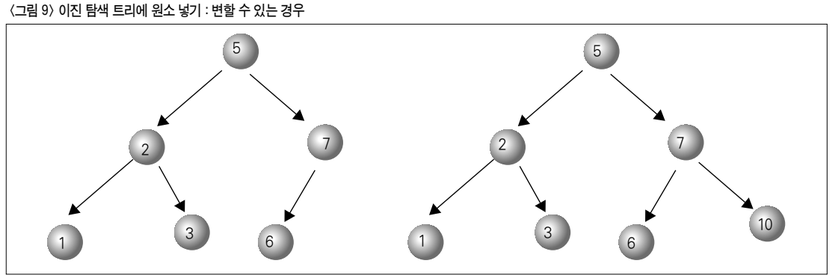
◆ **프로그램에서 예전의 집합은 더 이상 사용하지 않고 항상 최신의 집합만 사용하는 경우** : 이 경우는 값 중심의 프로그래밍 방식이나 그렇지 않은 방식이나 똑같은 비용이 든다. 이전 것은 더 이상 사용되지 않으므로, 현재의 집합위에 덧붙여서(destructive update) 새로운 원소를 매달면 그만이다. 드는 시간은, 새로운 원소의 위치를 찾는 시간 log N(트리 위의 한 경로) 만큼이다. 메모리는 새로 매다는 원소 하나만 더 있으면 된다.

구체적으로 따져보았듯이, 이진 탐색 트리로 구현된 집합에 새로운 원소를 첨가하는 연산을 구현할 때, 값 중심의 프로그래밍(값은 변하지 않는다)을 가정하면 시간과 메모리의 소모가 그렇지 않은 경우와 같거나 그보다 오히려 적게 된다.

〈그림 8〉 이진 탐색 트리에 원소 넣기 : 변하지 않는 경우



〈그림 9〉 이진 탐색 트리에 원소 넣기 : 변할 수 있는 경우



또 다른 '네오'가 나올 때

값 중심의 프로그래밍(value-oriented programming)은 기존의 물건 중심의 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 프로그래밍하는 복잡함 대신에, 프로그램은 값을 계산하고 정의하는 것뿐이다. 우리가 중고등학교 시절 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 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. 값 중심의 프로그램들은 실행 비용이 많이 드는 건 아닐까 염려할 필요는 없다. 전통적인 컴파일러 기술들이 어셈블리 프로그래밍을 만족스럽게 대체시켰듯이, 값 중심의 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 싶어 하는 영화 매트릭스의 네오들이 나설 무대는 준비되어 있다.

이쯤해서 이번 첫 회의 내용을 마치도록 하자. 다음호에서는, 이 연재에서 다루기로 한 nML이라는 값 중심의 프로그래밍 언어가 컴퓨터 소프트웨어의 발달과정에서 어느 위치에서 어떤 역할을 하도록 고안되고 있는지를 살펴보도록 하겠다. 이렇게 2회의 글을 통해 nML을 보는 우리의 안목을 준비한 후에, 3회부터는 구체적인 nML 프로그래밍의 세계로 여행해보도록 하자.

정리 : 강경수 elegy@sbmedia.co.kr