nML과 함께 하는 프로그래밍 여행 5

nML 언더그라운드 콘서트

강현구 hgkang@ropas.kaist.ac.kr
KAIST 프로그래밍언어 연구실/프로젝트 분석 시스템 연구단 박사과정에 재학 중이며, 데이터베이스 응용프로그램을 자동으로 최적화해 주는 시스템을 개발하고 있다.

이번 호에서 소개할 것은 독자들에게 조금 익숙하지 않은 내용일 수도 있다. 지난 호에서 비교적 단순한 기능을 살펴보았다면, 이번 호에서는 코드 재사용을 위한 조금 복잡하지만 흥미로운 기능을 소개한다.

지난 연재를 통해 nML에 대한 전반적인 소개를 마쳤다. 이번 호에서는 예제를 통해 nML 과 기존의 C, C++, 자바 등의 언어에서 다양한 프로그래밍 모델을 비교해 보겠다. 이번 호에 소개될 예제는 프로그래밍에 대한 우리의 생각의 틀을 다양하게 재조명해 줄 것이다. 이를 통해 프로그램을 바라보는 눈썰미를 보다 세련되게 다듬어 보자.

시작하기 전에 한 가지 당부 드리고 싶다. 지금부터 보여지는 nML 예제에 대해 독자들이 익숙한 프로그래밍 언어로 같은 프로그램을 작성하면 어떻게 되는지 생각하면서 읽으면 더욱 도움이 될 것이다.

메모리 중심 프로그래밍 vs. 값 중심 프로그래밍

우선 지난 연재에서 소개된 값 중심의(value-oriented) 프로그래밍 패러다임과 메모리 중심(imperative) 패러다임의 차이를 되짚어 보는 차원에서 리스트를 뒤집는 함수를 각각 C와 nML로 작성해보고, 그 차이에 대해 한 번 살펴보자.

먼저 C 언어로 작성된 간단한 함수가 하는 일을 이해하자. 몇 줄 안되는 코드이니 먼저 다음의 코드를 이해한 후에 다음의 해설로 넘어갔으면 한다.

```
struct datanode {
  int value;
  struct datanode *next;
};

struct datanode* rev (struct datanode *list)
{
  struct datanode *a, *b, *c;
  if (!list)
     return (NULL);
  else {
     a = NULL;
     b = list;
     c = b->next;
     while (c) {
       b->next = a;
       a = b;
       b = c;
       c = c->next;
     }
     b->next = a;
     return(b);
  }
}
```

그렇다. 이 코드는 C 언어로 작성된 프로그램에서 비교적 자주 나오는 패턴인 리스트를 뒤집어주는 함수다. 본 연재를 읽는 수준의 독자들이면 누구나 쉽게 이해했을 것이라 생각한다. 같은 일을 하는 nML 함수를 작성해보면 다음과 같이 된다.

```
fun rev [] = []
  | rev (hd::tl) = rev tl @ [hd]
```

어떤 차이가 느껴지는가? 우선 nML에서는 loop을 사용하지 않고 재귀호출(recursive call)을 사용하고 있다. 재귀성(recursion)은 우리가 데이터 구조나 계산을 바라보는 방법인, 귀납구조(induction)로 프로그램을 바라보게 만드는 가장 자연스런 '생각의 틀'임을 상기하자. 결국 nML에서는 '리스트를 뒤집는다'를 우리가 머리 속에서 생각하는 귀납적 사고 그대로(리스트의 맨 앞 원소를 나머지 리스트를 뒤집은 리스트의 뒤에 붙인다) 프로그램으로 표현할 수 있게 해준다.

반면, C 언어의 예제 스타일은 이와 같은 '리스트를 뒤집는다'라는 생각뿐만 아니라 물리적 메모리 구조에 맵핑해주는 작업까지 동시에 생각하게 하고 있다. 결국, 이와 같은 C 언어 스타일의 '생각의 틀'은 번거로울 뿐만 아니라 더 많은 버그의 가능성과 디버깅의 고통을 수반하게 된다.

여기서 혹자들(전형적인 C 프로그래머들)은 이와 같이 프로그래밍하면 컴파일러가 성능이 떨어지는 코드를 생성하지 않을까 염려한다. 그렇다. 아니 정확히 이야기하면 전산학의 원시 시대였던 C 언어가 나왔을 당시(1970~80년대)에는 그랬다. 하지만 그로부터 30여년이 지난 지금은 많은 발전이 이뤄져서 이처럼 하위 수준 쟁점(메모리로의 맵핑 구조를 고려해야 하는)들로부터 해방됐다. 충분히 요약된 관점에서 프로그래밍할 수 있으면서도 C 언어에 못지않은 성능을 가지는 실행 코드를 생성해주는 프로그래밍 언어들이 속속 나타나고 있는 추세다. 참고 자료 ⓫을 보면 프로그래밍 언어들 간의 성능을 비교하는 실제의 실험 자료들이 있다. nML의 가까운 친척 OCaml, SML 등이 상당히 좋은 성능을 보여주고 있음을 확인할 수 있다.

다형 데이터 구조 다루기

여기서는 코드의 재사용성을 높여주는 다형성(polymorphism, generic feature)을 잘 활용하는 방법과 그 의의에 대해 알아보겠다. 즉, 프로그래밍할 때 자주 다루게 되는 다양한 데이터 구조들(리스트, 트리, 맵, 집합, 큐, 스택, 힙 등)을 다형적으로 사용하는 방법론에 대해 이야기해 볼 것이다. 또한, 언어 차원에서 이러한 다형성을 제공하지 않는 C, 자바, C++ 등의 언어에서는 이런 경우를 어떻게 해결하려 하고 있는지 알아보겠다.

우선은 우리가 프로그래밍할 때 가장 자주 다루게 되는 데이터 구조인 리스트를 다형적으로 활용하는 예를 살펴보자.

```
# fun list_length([]) = 0
    | list_length(hd::tl) = 1+list_length(tl) ;;
val list_length: 'a list -> int = <fun>
```

list_length 함수는 임의의 리스트를 받아 그 길이를 결과로 주는 함수다. 함수의 입력 파라미터로 받는 리스트의 원소 타입이 특별히 고정되어 있지 않은 것(다형성)을 알 수 있다. 즉, 어떤 타입의 리스트를 인자로 줘도 list_length 함수는 잘 동작한다. 예를 들어 list_length [1,2,3] 식으로 정수 리스트를 인자로 줘도, list_length ['a', 'b', 'c'] 식으로 문자 리스트를 인자로 줘도 잘 동작한다.

C 언어에서는 프로그래밍 언어 차원의 리스트 데이터 타입과 다형 함수를 지원하지 않기 때문에 리스트의 원소의 타입에 따라서 각각 구조체 정의와 해당 list_length 함수의 정의를 다음과 같이 반복해줘야 한다.

```
struct int_list {
  int element;
  struct int_list* next;
}
struct char_list {
  char element;
  struct char_list* next;
}
int int_list_length(struct int_list* list) {
  if (list==NULL) return 0;
  else return (1 + int_list_length(list->next));
}
int char_list_length(struct char_list* list) {
  if (list==NULL) return 0;
  else return (1 + char_list_length(list->next));
}
```

여기 예제는 간단하지만, 다루는 리스트의 종류가 많아지고 해당 리스트에 대한 list_length 같은 operation들이 많아지면 그 효과는 더욱 커진다. 참고로 C와 C++에서는 이와 같은 경우 (void*)를 사용해서 다형성을 흉내낼 수 있다. 하지만 이 방법은 컴파일러가 타입 오류를 찾지 못하도록 하기 때문에 프로그래머가 직접 일일이 확인해줘야 하며, 수많은 버그를 양산하기도 하는 위험한 형태임을 명심해야 한다.

C++ 언어에서는 템플릿이라고 불리는 일종의 매크로 시스템을 사용해 재사용 가능한 알고리즘(클래스 혹은 메쏘드)을 '명시적으로' 정의해 사용할 수 있도록 도와주고 있다.

```
template <T> class List {
public:
  struct Node {
    T value;
    Node* next;
  }

  struct Node* head = NULL;

  void add(T value) {
     struct Node* n = (struct Node*)malloc(sizeof(struct
        Node));
     n->value = value;
     n->next = head;
     head = n;
  }

  int length() {
     struct Node* n;
     int i = 0;
     for(n = head; n != NULL; n = n->next) {
        i++;
     }
     return i;
  }
  ...
}
...
List<int> l;
l.add(1); l.add(2); l.add(3);
cout << l.length();
...
```

예제의 List〈T〉 클래스는 T 부분을 차후 원하는 타입으로 초기화(instantiation)하여 사용할 수 있다. 즉, 리스트의 원소 타입에 따라 해당 리스트 클래스를 따로 정의하지 않고, T 타입에 해당되는 부분만 정해줌으로써 사용할 수 있게 해준다. 그러나 이 방법은 C++의 발명자 Bjarne Stroustrup이 그의 저서 『The C++ Programming Language, 3rd』에서 언급하고 있듯이, 템플릿 부분에 대해서는 컴파일러가 미리 타입검사를 할 수 없고, 차후 실제로 템플릿을 초기화해 사용하는 시점에서야 비로소 타입 오류를 확인할 수 있다는 근본적인 문제점이 있다. 필자 개인적으로 템플릿은 다형성을 비교적 비슷하게 흉내내는 쓸만한 기능이라고 생각하지만 앞에서 지적한 것처럼 그리고 5회에 걸친 연재를 통해 계속 강조해 왔듯이 자칫 버그를 일으킬 구멍을 언어 차원에서 가지고 있기 때문에 미래의 모범답안이 아님을 간단히 지적하고자 한다.

자바 언어에서는 리스트 데이터 구조를 사용하기 위해 보통 추상 클래스인 AbstractList를 extend(구현)하여 사용하거나 Vector를 사용하는데, 여기선 Vector를 사용하는 상황을 기준으로 예제를 들겠다.

```
...
public int list_length(Vector v) {
public int length = 0;
  for (Enumeration e = v.elements() ; e.hasMoreElements() ;)
{
      length++;
  }
  return length;
}
...
Vector l1 = new Vector(3);
Vector l2 = new Vector(3);
l1.add(new Integer(1));
l1.add(new Integer(2));
l1.add(new Integer(3));
l2.add(new Character('1'));
l2.add(new Character('2'));
l2.add(new Character('3'));
...
```

리스트 길이를 재기 위해서는 Vector.size()를 호출하면 되지만, 여기서는 다형성을 이해하기 위해 미리 준비되어 있지 않은 새로운 함수를 정의하는 상황으로 가정했다(nML도 List.length가 미리 정의되어 있음). 앞의 코드를 보면 마치 다형적으로 add를 할 수 있는 것으로 보이는데, 자세히 보면 그 이유는 Vector. add 함수의 입력 파라미터가 Object 타입으로 선언되어 있기 때문에 어떤 객체도 전달할 수 있는 것이다. 따라서 구체적인 타입정보를 잃어버린 Object 모음이 되어 버리는 것이다. 이는 차후 리스트의 한 원소 값을 나중에 참조하려면 (Integer) l1.get(0) 식으로 동적 형변환(dynamic type casting)을 사용해서 프로그래머가 직접 원래의 구체적인 타입으로 바꿔야 하는 문제를 야기한다(C에서 void형 포인터를 사용하는 것과 거의 유사하다).

이는 불편할 뿐만 아니라, l1의 첫 번째 원소가 실제로 Integer 타입이 아닌 경우 실행 시간에 예외상황이 발생한다. 즉, 컴파일 시간에 오류를 미리 찾아낼 수 없다는 단점이 있다. 또한 실행시간 타입 검사로 인해 이와 같은 상황이 많을수록 성능이 떨어지게 된다. 이와 같은 문제 때문에 썬에서는 차후 버전의 자바 언어는 다형성(polymorphism 또는 generic feature)을 지원하도록 확장하겠다고 선언(Generic Java)했으며, 마이크로소프트 역시 닷넷 중간 언어의 다형성 지원을 위한 확장을 고려하고 있다고 한다.

마지막으로 자바, C++와 같은 객체지향 언어에서는 클래스 구조(class hierarchy)와 동적 바인딩(dynamic method dispatch)을 잘 활용해 다형성을 흉내낼 수 있다. 예를 들어 length라는 virtual(abstract) 메쏘드를 가지고 있는 List 추상 클래스를 상속해 IntList, StringList 등의 클래스를 사용자가 정의하여 length를 구현해 준다. 차후, List 타입에 속하는 모든 객체는 length를 그냥 호출해 사용할 수 있다.

하지만 이 방법은 아주 간단한 데이터 구조에 대해서도 여러 개의 클래스 정의와 같은 이름/의미를 가지는 virtual 메쏘드들의 정의를 여러 클래스에 분산시킨다. 따라서 간단한 다형 데이터 구조를 사용자가 정의하여 사용하고 싶을 경우에 쓸데없는 코드 크기 증가와 이로 인한 관리하기 어려움 등의 문제들을 야기한다. 또한 다형성을 언어차원에서 직접 지원하는 것이 아니고, 일종의 다형성을 모델링 해주는 패턴을 통해 해결하려는 차원이기 때문에 이를 '다형성을 지원한다'라는 의미로 확대시키기에는 문제가 있다고 본다.

지금까지 다형 리스트(nML에서의 'a list 타입)를 활용해 다양한 입력 인자 타입에 대해 재사용 가능한 함수를 정의/사용하는 방법을 예제를 통해 살펴보았다. 이제 시야를 넓혀서 좀더 거시적으로 바라보자. 다형성이 제공되면, 예제에서처럼 다형 트리구조, 다형 집합, 다형 맵, 다형 스택, 다형 힙 등의 데이터 구조를 활용하여 재사용 가능한 다형 함수(라이브러리/알고리즘/프로그램)들을 정의하여 사용할 수 있게 된다. 필자는 대학 시절 C 언어를 사용해 임의 타입의 원소를 가지는 리스트를 입력받아 정렬(heap sort)해 주는 일반형(generic) 알고리즘을 작성했던 시절을 어렴풋이 기억한다. 그때는 마치 커다란 프로그래밍 기법을 하나 배운 것처럼 기뻐했었다. 그런데 nML을 사용하고부터는 이런 일반적인 알고리즘을 설계하는 방법을 모두 잊어 버렸다. 그냥 된다.

고차함수와 다형성을 가지는 데이터 구조

실제로 다형 리스트와 같은 다형성을 가지는 데이터 구조(트리, 스택, 큐, 맵, 집합 등)는 함수가 값임(인자로 전달 혹은 리턴 가능)을 잘 활용했을 때, 재사용성과 같은 그 효과가 극대화된다. 예를 들어 다음의 프로그램을 생각해 보자.

```
# fun print_list printer sep list =
    case list of
      [] => print_newline()
    | hd::[] => printer hd; print_newline()
    | hd::tl => printer hd; print_string sep; print_list
       printer sep tl ;;
val print_list : ('a -> 'b) -> string -> 'a list -> unit =
<fun>
```

print_list 함수는 첫 번째 인자로 전달받은 printer 함수를 사용해 리스트의 각 원소를 프린트해 준다. 그 의미를 좀더 자세히 살펴보자. 먼저 print_list 함수의 타입은 ('a -> 'b) -> string -> 'a list -> unit으로 자동 유추되었다. ('a -> 'b) 타입으로 유추된 첫 번째 인자 printer는 리스트의 원소를 프린트해 주는 함수다. string 타입으로 유추된 두 번째 인자 sep는 프린트시 원소 사이의 분리자(separator)다. 'a list 타입으로 유추된 세 번째 인자는 프린트할 리스트다. 이제 print_list를 사용하는 예를 살펴보자.

```
# print_list print_int ", " [1,2,3] ;;
1, 2, 3
val it: unit = ()
# print_list print_char  " | " ['a','b','c'] ;;
a | b | c
val it: unit = ()
```

우선, [1,2,3]이라는 int list로도 ['a','b','c']라는 char list로도 print_list 함수를 호출해 사용할 수 있음을 확인할 수 있다(다형성). 두 번째로 리스트 원소의 타입에 따라 적절한 프린터 함수

를 파라미터로 전달하여 활용하고 있음을 볼 수 있다(고차 함수 : higher-order function).

이와 같은 스타일은 처음 보는 사람에게는 조금은 생소하게 느껴질 것이다. 어쩌면, 처음의 필자처럼 약간 감동했을 수도 있다. 그런데 이런 경우가 정말 자주 있나라는 의문이 바로 든다. 필자도 처음 실제 프로그래밍할 때 이렇게 함수를 값으로 전달하여 사용하는 경우가 얼마나 있을지 의문이 많았다(여기서 잠시 멈추고 자신의 경험에 비추어 한 번 생각해 보길 바란다). 하지만 반복되는 경험을 통해 GoF의 『디자인 패턴』에서 나오는 패턴만큼이나 혹은 일부 경우는 디자인 패턴 자체(차후 자세히 거론)로서 빈번하게 사용되고 있음을 알게 되었다. 그런데 기존의 언어들에서는 언어 차원에서 고차 함수를 지원하지 않기 때문에 일종의 고차 함수 흉내내기를 통해 이런 패턴들이 나타나게 된다. 아직 감이 잘 안 오는 독자들이 많을 것 같다. nML에서 고차 함수를 잘 활용하는 방법에 대해서는 일단 미뤄두고, 우선은 기존의 언어에서의 고차 함수 흉내내기 패턴들을 한번 살펴보자.

자바에서 고차 함수 흉내내기

자바의 GUI 프레임워크인 AWT, Swing 라이브러리는 이벤트 위임 모델(Delegation Event Model)이라고 불리우는 이벤트 처리 메커니즘을 디자인해 사용하고 있다. 이벤트 위임 모델이란 다음의 예제처럼 임의의 GUI 컴포넌트에 어떤 이벤트가 발생했을 때 해야 하는 일을 구현하는 함수를 가지고 있는 객체를 프로그래머가 해당 컴포넌트에 이벤트 핸들러(listener)로서 등록시켜 주는 식의 모델을 말한다.

```
public class EventHandlingExample extend JFrame {

  public JButton myButton = new JButton();

  public class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
       // 이벤트 발생시 해야할 일 구현
    }
  }

  public static void main(String [] args) {
    ...
    // myButton에 이벤트 발생시 해야할 일 등록
    myButton.addActionListener(new ButtonListener());
     ...
  }
}
```

앞의 코드를 자세히 살펴보면, 의미상 myButton.addActionListener 함수의 인자로 전달하고 싶었던 내용은 단지 버튼이 눌렸을 때 해야 하는 일(ActionEvent 타입의 인자를 받아서 void를 결과로 주는)을 뜻하는 함수 하나였음을 알 수 있다. 차후 버튼 컴포넌트는 등록된 핸들러를 사용해(delegation) Button Click 이벤트를 처리한다. 그런데, 자바에서는 함수를 직접 값으로 전달할 수 없기 때문에, 해당 함수를 'actionPerformed'라는 특정 이름으로서 구현하여 가지고 있는 ActionListener 인터페이스 타입의 객체를 인자로 주는 방식을 사용하고 있음을 알 수 있다.

nML 스타일로 위의 ActionListener를 등록하는 부분을 다시 표현하면 다음과 같이 간단히 된다.

```
structure Button = NButton()  // 새로운 버튼 모듈 생성
Button.addActionListener(fn e => (* 이벤트 발생시 해야할 일 구현
*))
```

이와 같이 함수를 인자로 주는 상황을 해당 함수를 (특정 이름으로) 가진 객체를 인자로 주기를 통해 간접적으로 해결하는 형태는 FileFilter, FilenameFilter 등 자바의 기본 라이브러리 디자인의 곳곳에 묻어 있다.

콜백 함수는 고차 함수 흉내내기?

콜백(callback) 함수는 프로그래머라면 모르는 사람이 없을 정도로 너무나도 많은 프로그래밍 및 실행 환경에서 다양한 형태로 사용되고 있다. 일반적으로 콜백 함수라 하면 실행되는 프로그램 자신이 아닌, 프로그램이 실행되는 실행환경(OS, VM, 프레임워크 등)이 프로그래머가 의도(등록)한 특정 시점에서 호출하는 함수를 말한다. 즉, 프로그램의 실행환경이 할 일을 뜻한다. 이런 의미에서는 앞의 자바 예제도 일종의 콜백 함수라 할 수 있다.

```
#include <windows.h>

int  APIENTRY  WinMain(HINSTANCE  hInstance,HINSTANCE
hPrevInstance ,LPSTR lpszCmdParam,int nCmdShow)
{
  ...
}

void CALLBACK TimerProc(HWND hWnd, UINT uMsg, UINT idEvent,
DWORD dwTime)
{
  // my job
}

LRESULT CALLBACK WndProc(HWND hWnd,UINT iMessage,WPARAM
wParam,LPARAM lParam)
{
  switch(iMessage) {
       case WM_CREATE:  SetTimer(hWnd, 1,
       100, (TIMERPROC)TimerProc);  return 0;
       case WM_DESTROY: KillTimer(hWnd, 1);  PostQuitMessage(0); return 0;
  }
  return(DefWindowProc(hWnd,iMessage,wParam,lParam));
}
```

이 프로그램은 일정시간(0.1초) 간격으로 TimerProc 함수를 호출하는 윈도우 응용 프로그램의 뼈대다. WndProc, TimerProc은 사용자가 지정(요청)한 시점에 윈도우 운영체제가 호출하는 함수다. WndProc은 프로그램을 처음 시작할 때, TimerProc은 SetTimer 함수를 사용하여 프로그래머가 지정한 시간마다 운영체제에 의해 호출된다.

지금까지 고차 함수에 대한 이야기와 앞의 예제를 잘 이해한 독자들은 필자가 하려는 이야기를 이미 파악했을 것이다. 앞의 SetTimer 함수는 자바 예제에서의 addActionListener가 하는 일과 같은 일을 하고 있다. 즉, 운영체제가 특정 이벤트 발생시점마다 해야 하는(호출) 일(함수)들의 목록에 하나 더 추가하는 것이 그 일이다. 만약 윈도우 운영체제가 nML과 같이 고차 함수를 지원하는 언어로 다시 쓰인다면 특별히 Callback이라고 하는 새로운 개념(패턴)을 도입하지 않고, 그냥 다음과 같은 API만 있었을 것이라 감히 예측해 본다.

```
fun my_job_to_do (...) = ... // my job
Window.setTimer(my_job_to_do)
```

지금까지 이야기한 상황 말고도 다양한 프로그래밍 환경에서 콜백 함수, 이벤트 핸들러, 인터럽트 핸들러, 함수포인터 등의 여러 가지 이름으로 고차 함수 흉내내기 기법은 계속되고 있다.

지금까지 이야기를 정리해보자. 고차 함수의 한 형태인 함수를 인자로 주는 상황은 프로그래밍시 혹은 프레임워크 설계시 자주 활용되는 상황으로, 프로그래밍 언어 차원에서 지원되면 코드 라인수가 훨씬 줄어들고, 그 의미가 훨씬 명확해진다. 필자는 앞의 상황들을 합쳐서 고차 함수 흉내내기 패턴이라고 부른다.

이제 다시 본론으로 돌아가서 함수를 값으로 취급하는 nML에서 이를 충분히 활용하는 프로그래밍 예제들을 살펴보도록 하자.

map, iter, find, fold_left

제목에 표시된 네 가지 함수는 고차 함수와 다형타입 데이터 구조를 동시에 사용하면서 정의되는 nML 프로그래밍 패턴의 대표 주자들이니 잘 알아두면 좋다. 우선 List 데이터 구조에 대한 이 네 가지 함수의 사용 예를 살펴보자.

```
# List.map (fn n => sting_of_int n) [1,2,3] ;;
val it: int list = ["1","2","3"]
# List.iter (fn s => print_string s) ["n","M","L\n"];;
nML
val it: unit = ()
# List.find (fn n => n < 10) [20,13,9,3,30];;
val it: int = 9
# List.fold_left (+) 0 [1,2,3] ;;
val it: int = 6
```

List.map 함수는 리스트와 리스트의 원소에 취할 어떤 함수를 입력받아 각 원소에 해당 함수를 모두 적용한 결과 리스트를 준다. List.iter 함수는 리스트의 각 원소에 인자로 전달받은 함수를 모두 적용한다(맵과 달리 리턴되는 값이 없음). iter 함수 사용은 C에서는 리스트 데이터 구조에 대한 포인터를 이용한 loop, 자바에서는 Enumeration(iterator pattern) 인터페이스를 이용한 loop에 대응되는 것으로 이해하면 쉬울 것이다. List.find 함수는 입력 리스트를 검색해서, 인자로 받은 함수(predicate)의 결과를 true로 만드는 첫 번째 원소를 리턴해 준다. List.fold_left 함수는 먼저 두 번째 인자로 받은 값과 세 번째 인자로 넘겨받은 리스트의 첫 번째 원소를 첫 번째 인자로 받은 함수에 적용하고, 다시 그 결과를 리스트의 두 번째 원소에 대해서 적용하고, 다시 세 번째에 대해서… 식으로 모두 적용한 결과를 준다. 즉, 예제의 List.fold_left는 (+ (+ (+ 0 1) 2) 3)의 의미로 해석된다.

C 언어에 익숙한 독자들은 다음과 같이 다시 풀어 쓴 예제를 보는 것이 이해에 더 도움이 될 것 같다.

```
struct list {
  int elm;
  struct list* next;
}
int fold_left (int acc, struct list* l) {
  struct list* ptr;
  for (ptr = l; ptr != NULL; ptr = ptr->next) {
    acc = acc + ptr->elm;
  }
  return acc;
}
```

fold_left를 제대로 이해하기 위해 한 가지 더 연습해보자. 다음은 정수 리스트를 입력받아 이에 해당하는 10진수를 생성해 주는 예제다.

```
# List.fold_left (fn x y => 10 * x + y) 0 [1,2,3] ;;
val it: int = 123
```

앞의 네 가지 함수는 리스트 외에도 트리, 맵, 집합, 스택, 힙 등 다양한 데이터 구조에 대해서 다형적으로 정의되어 사용되므로 예제에서의 리스트나 주어진 간단한 원소 타입에 국한하지 말고 확장해서 이해하도록 하자. 앞에서 fold_left를 C로 풀어 쓴 것을 보았을 때 몇몇 독자들은 이미 눈치챘겠지만, 메모리 중심 언어의 대표적인 loop 구조를 함축적으로 요약해 놓은 함수들임을 숙지하고 있으면 좋다. 또한, 자바, C++ 등의 객체지향 언어에서 Collection 혹은 Container 객체들에 대해서 Enumerator 혹은 Iterator라고 불리우는 것들과 다형성을 합쳐 높은 개념이 이렇게 간단명료하게 표현되고 있다. 『디자인 패턴』의 'Behavioral Pattern'에 기술되어 있는 여러 가지 패턴들이 이런 식으로 고차 함수를 이용해 간결하게 재해석될 수 있다.

함수를 리턴해주는 함수

nML에서 t1 -> t2의 의미는 t1 타입의 인자를 받아 t2 타입의 값을 리턴해주는 함수였다. 그런데 앞서 예제로 살펴본, print_list 함수의 타입을 다시 자세히 살펴보면 ('a -> unit) -> string -> 'a list -> unit 타입으로 되어 있다. 즉, print_list 함수는 ('a -> unit) 타입의 인자를 받아서 (string -> 'a list -> unit) 타입의 함수를 리턴하는 함수인 것이다. 또한 리턴된 함수는 다시 string 타입의 인자를 받아서 ('a list -> unit) 타입의 함수를 준다. 이와 같은 이해를 바탕으로 다음의 예를 살펴보자.

```
val _ = print_list print_int "," [1,2,3]

val print_int_list = print_list print_int
val print_int_list_with_comma_sep = print_int_list ","
val _ = print_int_list_with_comma_sep [1,2,3]
```

결국 첫 번째 함수 호출은 다음 소스의 세 번의 함수 호출을 합쳐놓은 의미이다. 쉽게 생각하면 print_list를 항상 첫 번째 라인과 같이 사용하면 될 것 같은데, 2~4 라인의 경우처럼 부분적으로 호출할 필요가 있을까? 다양한 경우를 생각할 수 있지만, 필자는 print_list와 같이 일반적인 함수정의로부터 좀더 구체적인 함수를 재정의하여 사용하기 위해 주로 사용한다. 즉, 두 번째 라인처럼 print_list로부터 print_int_list라는 함수를 만들어서 사용하고 싶을 때 이와 같은 스타일을 사용한다. 다음 예제를 살펴보자.

```
# val find_positive = List.filter (fn n => n > 0) ;;
val find_positive: int list -> int list =
# find_positive [1,-3,4];;
val it: int list = [1, 4]
```

List 라이브러리의 filter 함수는 predicate 함수와 리스트를 인자로 전달받아서 해당 predicate를 만족하는 리스트를 리턴해주는 ('a -> bool) -> 'a list -> 'a list 타입의 함수다. 이와 같이 사용하면 정수 리스트를 입력받아 양수들만 뽑아낸 리스트를 그 결과로 주는 함수를 쉽게 정의해 사용할 수 있다.

이와 같은 이해를 바탕으로 다시 상황을 확장해보자. 어떤 식으로 함수를 리턴해 주는 함수를 잘 활용할 수 있을까? 예를 들어 클라이언트/서버 프로그래밍을 생각해 보자. 클라이언트가 서버에 원하는 함수를 요청하고, 서버는 이를 리턴해 주는 모델을 생각할 수 있다. 즉, 어떤 응용 프로그램의 어떤 버튼을 눌렀을 때, 수행해야 하는 일(함수)을 서버에서 얻어와(리턴) 실행하는 모델(자동 업그레이드)을 언어 차원에서 자연스럽게 구현할 수 있겠다.

지역 함수 선언

함수가 값이라는 사실에서 이미 눈치챘겠지만, 함수를 선언하는 도중에 필요시 지역 함수를 선언하여 사용할 수 있다.

```
fun tail_reverse list =
let
  fun trev list acc = case list of
      [] => acc
    | hd::tl => trev tl (hd::acc)
in
  trev list []
end
```

tail_reverse는 임의의 리스트를 뒤집어 주는 함수다. tail_reverse 함수의 몸통(body)을 보면 trev라는 지역(local) 함수를 정의해 사용하고 있음을 알 수 있다. 지역 함수는 지역 변수와 같은 용도로 사용된다. 그러나 함수를 특별한 것으로 취급하는 언어들에서는 의미상 지역 함수에 해당해도 이를 언어 차원에서 지원하지 않기 때문에, 어쩔 수 없이 현재 함수의 바깥쪽에 따로 선언해서 사용해야 한다. 하지만 nML에서는 이와 같이 자연스럽게 지역 함수를 정의해 사용할 수 있다.

의미상 지역 변수에 해당하는 것을 전역 변수로 선언해서 사용하면, 관련이 많은 코드들이 분산되어 읽거나 관리하기 어려워진다. 이는 결국 찾아내기 어려운 버그로 이어지는 경우가 있다. 결론적으로, 지역 함수의 적절한 사용은 프로그램을 읽기 쉽고, 관리하기 쉽게 해 줄 뿐만 아니라 버그없는 프로그램을 작성하게 해 주는 좋은 스타일이다.

복잡한 데이터 구조 다루기

nML과 같은 값 중심의 프로그래밍 언어를 사용해 본 대부분의 사람들이 빼놓지 않고 이야기하는 장점은 복잡한 데이터 구조를 쉽게 다룰 수 있다는 점이다. 잘 알려진 데이터구조 혹은 알고리즘 문제인 이진 검색 트리(binary search tree) 프로그램을 한 번 nML로 작성해 보자.

```
// filename : btree.n

exception EmptyTree  // exception 선언
type 'a btree = Leaf | Node of 'a * 'a btree * 'a btree
// 이진 검색트리 선언

// lookup : 'a btree -> 'a -> bool
// 이진 검색트리를 뒤져서 입력 키가 있는지 검사
fun lookup tree key = case tree of
      Leaf => false
    | Node(value,left,right) =>
      if (key < value) then lookup left key
      else if (value < key) then lookup right key
      else true

// insert : 'a btree -> 'a -> 'a btree
// 이진 검색트리에 입력 키 삽입
fun insert tree key = case tree of
      Leaf => Node(key,Leaf,Leaf)
    | Node(value,left,right) =>
      if (key < value) then Node(value,insert left key,right)
      else if (value < key) then Node(value,left,insert right
       key)
      else tree

// delete : 'a btree -> 'a -> 'a btree
// 이진 검색트리에서 입력 키 삭제(만약 없으면 예외상황 EmptyTree 발생
fun delete tree key =
let
  fun deletemin tree = case tree of
      Leaf => raise EmptyTree
    | Node(value,Leaf,right) => (value,right)
    | Node(value,left,right) =>
                let val (y,l) = deletemin left
                in (y,Node(value,l,right)) end
in
  case tree of
    Leaf => Leaf
  | Node(value,left,right) =>
    if (key < value) then Node(value,delete left key,right)
    else if (value < key) then Node (value,left,delete right
        key)
    else (
      case (left,right) of
        (Leaf,r) => r
      | (l,Leaf) => l
      | (l,r) => let val (z,r1) = deletemin r
                 in Node(z,l,r1) end
    )
end

// print : ('a -> 'b) -> 'a btree -> 'a -> 'a btree
// 이진 검색트리 프린트
fun print printer tree =
let
  val sp = fn () => print_string " "
in
  case tree of
    Leaf => ()
  | Node(v,l,r) => print printer l; sp(); printer v; sp();
    print printer r
end

val 현재 언어트리 = List.fold_left insert Leaf ["nML","Java","
    C++","C"]
val _ = print print_string 현재 언어트리; print_newline()
val 새 언어트리 = List.fold_left delete current_pl ["C","C++"]
val _ = print print_string 새 언어트리; print_newline()
```

우선 C나 자바로 이와 똑같은 프로그램을 다시 한번 작성해 보고, 다음의 해설을 읽기 바란다. 혹은, 책장에 『Data Structure in C (C++, Java)』 등의 책이 있다면 가져다가 프로그램의 각 부분을 비교해 보자.

데이터 구조를 정의하는 방법

세 번째 라인은 이진 검색트리를 다형 타입을 사용해서 정의하고 있다. 여기서 주지할 것은 원하는 데이터 구조를 선언하는 방법이 우리가 머리 속에서 이진 검색트리를 귀납적으로 정의하는 방법 그대로 쓰고 있다는 것이다. 즉, 이 라인의 의미는 다음과 같다. 임의의 타입의 원소를 가지는('a) 이진 검색트리(btree)는 다음과 같이 정의된다.

- 단말 이진 검색트리는 Leaf이다.
- 노드 이진 검색트리는 'a 타입의 원소 하나, 좌측 이진 검색트리 자식 그리고 우측 이진 검색트리 자식을 가진다.

데이터 구조를 만드는 방법

insert나 delete 함수를 보면 이진 검색트리 타입의 데이터를 생성하는 방법을 볼 수 있다. 여기서도 그냥 정의에 따라 단말 이진

검색트리면 Leaf라고 써주고, 노드 이진 검색트리면 Node(원소, 좌측자식, 우측자식) 식으로 써주면 데이터가 생성된다. C처럼 'malloc 등을 사용해 이진 검색트리를 위한 메모리를 해당 타입 크기만큼 할당해서' 라는 걸 전혀 생각하지 않아도 된다. 필요한 메모리는 nML 컴파일러가 알아서 할당해 안전하게(메모리 누수, 둥강난(dangling) 포인터는 전혀 걱정하지 않아도 됨) 관리해준다. 이 과정에서 쓸모없어진 메모리가 있으면 알아서 삭제해 가며 효율적으로 관리해 준다. 메모리 자동 관리(수거)에 대해 더 궁금한 독자들은 메모리 재활용기(garbage collector)에 관하여 찾아보기 바란다.

데이터 구조를 뒤지는 방법

lookup, insert, delete 등의 함수를 보면 이미 지난 연재에 이미 소개됐던 패턴 매치를 통해 데이터 구조 내에서 필요한 데이터를 꺼내 쓸 수 있는 것을 볼 수 있다. 여기서도 데이터 구조의 귀납적 정의 그대로 'Leaf이면 A를(Leaf => A), 노드이면 B를(Node(v,l,r) => B) 한다' 식으로 우리가 정의에 대해 생각하는 그대로 쓸 수 있다(참고로 디자인 패턴의 Visitor 패턴이 nML의 패턴 매치와 비슷한 의미를 가진다).

데이터 구조 관리

C나 C++에 익숙한 독자들은 앞의 함수 정의를 보면서 몇 가지 의문점이 있었을지 모르겠다. 예를 들어 함수에 데이터 구조가 전달될 때 값에 의한 호출(call by value)인가? 아니면 참조에 의한 호출(call by reference)인가? 그 외에도 C/C++ 프로그래밍의 관점에서 보면 수많은 의문이 생긴다. 해당 함수가 리턴하는 트리 구조와 인자로 전달받은 트리구조 사이에 메모리 공유가 있을 수 있나? 아니면 새로 할당해 복사하여 돌려주게 되나? 데이터는 언제 삭제되지? 만약 공유가 있다면 동강난(dangling) 포인터가 안 나도록 어떤 식으로 삭제하지? C/C++의 관성대로라면 다루고 있는 데이터 구조들에 대해 알고리즘 자체와 함께 수많은 생각을 하며 구현하던 상황을 비교하게 될 것이다. nML에서는 지금까지 나열한 모든 문제는 컴파일러가 알아서 다 관리해준다.

혹시, 데이터 구조에 대한 메모리로의 표현을 프로그래머가 직접 관리하지 못하기 때문에 효율성이 떨어지지 않을까 걱정이 된다면 앞서 소개한 성능비교 홈페이지를 참고하기 바란다. 결론적으로 nML의 간편한 데이터구조 처리 방법은 프로그래머들을 메모리 관리와 같은 하위 레벨의 쟁점들로부터 자유롭게 만들어서, 알고리즘과 같은 본연의 문제에 집중할 수 있도록 도와준다.

nML은 간단하다

아직 감이 잘 안 잡히는 부분이 있을지도 모르겠다. 하지만 이번 호까지 잘 따라 왔다면 독자들은 이미 nML 언어를 대부분 이해했다고 해도 과언이 아니다. nML의 언어 설명서는 단 43페이지로 이뤄져 있다. Arnold & Gosling의 자바 언어 설명서는 첫 번째 출판이 약 300페이지, Bjarne Stroustrup의 C++ 언어 설명서는 라이브러리에 대한 부분을 빼도 약 500페이지 이상이다.

한마디로 말해 컴파일러가 똑똑해졌기 때문에 배울 것이 줄어든 것이다. 가장 양이 많은 C++ 언어 설명서를 읽다 보면 실제로는 자바나 nML에서 컴파일러가 자동으로 해주는 일들을 프로그래머들이 직접 해 줄 것을 요구하고 있다. 예를 들어 '메모리 관리는 이렇게 해줘야 한다' 또는 'C++의 타입 시스템이 이런 타입 오류는 감지 못하니 조심해서 써야 한다' 등 지면의 많은 부분을 이런 설명을 위해 사용하고 있다. nML에서는 그런 문제를 컴파일러가 다 알아서 처리해 주기 때문에 프로그래머는 걱정할 필요도 알 필요도 없다(물론 컴파일러 개발자는 알아야 한다).

표현력에 있어서도 문제가 되지 않는다. 오히려 다형성, 고차 함수 활용과 같은 그 표현력이 더 높아졌다. 사실 표현력은 오히려 앞서지만, 기존 운영체제나 프레임워크가 C를 기반으로 작성되어 있기 때문에, 시스템 프로그래밍 영역을 위해 어쩔 수 없이 복잡한 C나 C++를 배울 수밖에 없는 것이라고 말하고 싶다.

Effective C++의 재해석

C나 C++는 지금까지 강조해왔던 세련된 컴파일러 기술(안전한 타입시스템, 자동 메모리 관리 등)을 제대로 갖추지 못하고 있다. 결국 이는 프로그래밍 언어의 설명서로 아니면 좋은 프로그래밍 스타일을 전해주는 Scott Meyers의 『Effective C++』와 같은 책의 몇몇 항목으로 혹은 GoF의 『디자인 패턴』과 같은 책의 몇몇 패턴을 통해 프로그래머의 몫으로 떠 넘겨지는 경우가 있다.

예를 들어 똑똑한 포인터(smart pointer)에 대해 알아보자. 똑똑한 포인터(smart pointer)란 프록시 패턴(Proxy Pattern)의 일종으로 C++ 언어에서 문제를 일으키곤 하는 포인터를 직접 사용하는 대신에 사용하고 싶은 객체를 담을 그릇을 미리 마련해 여기다가 원하는 객체를 담아서, 마치 포인터인 것처럼 사용할 수 있는 연산자를 제공하고, 메모리 초기화, 해제, 동강난(dangling) 포인터 등의 관리는 자동으로 되도록 만들어둔 것이다. 설명이 좀 복잡했는데, 참고 자료의 OO Tips 페이지에 가보면 똑똑한 포인터를 잘 설명해주고 있다. 조금 어려울지도 모르겠다. 하지만 nML(혹은 자바, C#)에서는 더 이상 걱정할 필요가 없어졌다. 컴파일러가 알아서 다 관리해 준다.

결론적으로 nML, 자바, C# 등과 같이 자동으로 메모리를 관리해 주는 언어에서는 Effective C++(1996년판) 책의 항목 5번부터 항목 17번까지의 내용은 더 이상 논할 필요가 없어진다. 또한 nML, 자바와 같이 컴파일할 때 타입오류를 완벽하게 검출해주는 언어로 인해 항목 46번은 의미가 없어졌다. 그 외에도 많은 항목들의 미묘한 부분 부분이 실제로는 컴파일러가 해줘야 하는 일을 못해주기 때문에 프로그래머의 몫으로 돌리고 있다는 사실을 알아둘 필요가 있다. 컴파일러 기술이 발전할수록 'Effective xx언어 프로그래밍' 과 같은 책은 얇아진다.

nML로도 쓸만한 응용 프로그램을 만들 수 있나요?

제목과 같은 질문을 필자는 몇 번 들어 본적이 있다. 아마도 생소한 프로그래밍 언어를 배우기 전에 그 가치를 확인하고 싶을 것이다. 필자는 이 질문에 '그렇다' 혹은 '아니다' 라고 직접적으로 답변해 주기보다는 다음과 같이 말하고 싶다.

nML과 같은 값 중심 프로그래밍 언어는 지금까지 계속 강조해 왔듯이 과거에 나온 어떤 다른 언어보다도 간단해서 배우기 쉽고, 표현력이 높으며, '안전한(버그 없는)' 프로그램을 작성할 수 있게 해주는 프로그래밍 언어다. 또한 그 우수성으로 인해 최근엔 마이크로소프트(이하 MS)에서도 C#을 이을 수 있는 차세대 언어로서 활발하게 연구/개발하고 있는 상태에 있는 언어다(이에 대한 자세한 정보는 참고자료의 MS 연구소의 연구 주제 페이지에서 'Programming Tools & Techniques' 란 주제 밑의 링크들을 차근차근 살펴보길 바란다).

사실 nML과 같은 값 중심 언어의 개념과 연구가 시작된 것은 벌써 20여년이 넘었고, 기술적인 어려움 때문에 천천히 발전해서 그 결과물들이 눈에 보이는 현실로 나타나기 시작한 것은 실로 최근의 일이다. 그리고 이제는 MS 역시 이 '뜨거운 감자'를 먼저 잡기 위해 뛰어들었다. 자랑할 점은 우리도 프로그래밍 언어계의 이 '뜨거운 감자'를 이미 쥐고 있다는 점이다. 하지만 아직까지 우리의 nML 언어는 자바의 AWT, Swing, EJB 혹은 MS의 Win32 API, MFC, COM+와 같이 윈도우용 응용 프로그램을 작성하기 위한 프레임워크를 가지고 있지 못하다. 또한 MS의 비주얼 C++와 같이 세련된 개발 도구와 개발 프레임워크도 가지고 있지 못하다. 아쉬운 점은 이처럼 세련된 프로그래밍 환경을 구축하는 문제에 있어 앞으로 마이크로소프트의 발걸음을 따라가기 벅찰지도 모른다는 안타까움이다.

현재 nML은 지난 3년간의 카이스트 전산과 프로그래밍 언어 코스를 통해 어느 정도의 사용자를 가지고 있다고 할 수 있다. 이번 연재를 통해 국내에 nML 사용자 저변이 조금이라도 더 확대된다면, 우리의 발걸음이 더 빨라질 수 있다는 역설적인 답변으로 질문에 대해 응답하고자 한다.

백문이 불여일견

앞서 이야기들에서 각 언어들을 비교할 때, nML에 편중되는 관점으로 생각의 틀을 집중시킨 데 조금은 오해가 있었을 지도 모르겠다. 그래서 필자의 의도를 다시 정리해 보고자 한다. 주어진 문제를 풀기 위해서는 문제의 특성에 따라 알맞은 옷을 입고 접근해야 한다. 알맞은 옷을 선택할 줄 아는 세련된 안목을 위해 이번 호에서는 nML이라는 새로운 생각의 틀을 중심으로 기존 생각의 관성을 다각도로 재조명했다.

백문이 불여일견(전산용어로는 '百見이 不如一打' 라고도 함)이란 말이 있다. 관심이 있는 독자는 nML 시스템을 설치해서 새로운 생각의 틀에서 한번 뒹굴어 보기를 권장한다. 독자 여러분의 생각의 틀 발전을 위해서 그리고 우리나라 IT 산업의 발전을 위해서 말이다. 마침내 5회에 걸친 긴 연재에 마침표를 찍을 시간이다. 그동안 관심있게 읽어 주신 독자들께 진심으로 감사드린다.

정리 : 강경수 elegy@sbmedia.co.kr

참고 자료
[1] The nML Programming Language System, Research On Program Analysis System, KAIST, http://ropas.kaist.ac.kr/n
[2] The Standard ML of New Jersey, Lucent Technologies, http://cm.bell-labs.com/cm/cs/what/smlnj
[3] The Object Caml Language System, Institut National de Recherche en Informatique et en Automatique, France, http://caml.inria.fr
[4] SML.NET, Microsoft Research at Cambridge http://research.microsoft.com/projects/sml.net/
[5] Introduction to Programming Using SML, Michael R. Hansen and Hans Rischel, Addison-Wesley, 1999.
[6] The Little MLer, Matthias Felleisen and Daniel P. Friedman, MIT Press, 1998
[7] The Functional Approach to Programming, G. Cousineau and M. Mauny, Cambridge University Press, 1998.
[8] Elements of ML Programming, Jeffrey D. Ullman, MIT Press, 1997.
[9] ML for the Working Programmer, Lawrence C. Paulson, Cambridge University Press, 1996.
[10] Abstract Data Types in Standard ML, Rachel Harrison, John Wiley & Sons, 1993.
[11] Computer Language Shootout Scorecard by Doug Bagley, http://www.bagley.org/~doug/shootout/craps.shtml
[12] Smart Pointer, OO Tips page, http://ootips.org/yonat/4dev/smart-pointers.html
[13] Microsoft Current Research, Microsoft Research http://research.microsoft.com/research