nML과 함께하는 프로그래밍 여행 4

nML 프로그래밍의 쓰임새 II

이욱세 cookcu@ropas.kaist.ac.kr
KAIST 프로그래밍언어 연구실/프로그램 분석 시스템 연구단 박사과정에 재학중이며, nML 컴파일러 개발자이다.

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.

편집자 주
지난 호에서 편집 과정상 소스가 잘못 표기된 부분이 있었습니다. 지난 호의 소스 중 '… gcd (n ;' 라고 표시된 부분을 '…gcd (n % m) m;;' 으로 바로 잡습니다.

연재순서
1회 2002.6 | 다른 생각의 틀, 값 중심의 프로그래밍
2회 2002.7 | 소프트웨어 기술의 발달 과정과 nML의 역할
3회 2002.8 | nML 프로그래밍의 쓰임새 I
4회 2002.9 | nML 프로그래밍의 쓰임새 II
5회 | nML 언더그라운드 콘서트

지난 호에서는 기본적인 타입, 함수, 사용자 정의 타입들, 예외 상황을 다루었다. 이번 호에서는 코드의 재사용에 유용한 다형 타입(polymorphic type), 함수를 주고받는 함수(higher-order function) 그리고 모듈에 관련된 내용을 소개하겠다. 조금 익숙하지 않더라도 코드 재사용을 위한 매우 흥미로운 기능이니 독자 여러분도 참을성 있게 따라와 주기 바란다.

본 내용에 들어가기 전에 준비 운동하는 기분으로 우선 기존 언어(C, C++, 자바 등)의 관성을 그대로 담아낸 기능을 살펴보자. 즉, 변경 가능한 메모리 장소, 배열, 루프 등을 먼저 소개하겠다.

메모리 반응과 루프

기존의 언어에 익숙한 사용자가 당장 nML로 프로그래밍하려고 하면 어려움을 느끼게 된다. 프로그래밍 패러다임이 다르기 때문이다. 앞서 소개된 것처럼 nML은 수학적 함수를 기반으로 하고 있다. 하지만 기존 언어들은 메모리 반응을 기반으로 하고 있다. 예를 들어, C로는 factorial 함수를 일반적으로 다음과 같이 작성할 것이다.

```
int factorial(n)
int n;
{
  int i;
  int acc = 1;
  for(i=1; i<=n; i++) {
    acc = acc * i;
  };
  return acc;
}
```

우선 증가 변수 i와 누적 변수 acc를 준비해 i를 증가시키면서 acc에 곱해 나가는 것이다. 그러나 nML로는 많이 다르게 작성한다.

```
fun factorial n =
  if n=0 then 1 else n * factorial(n-1)
```

factorial 5를 계산하기 위해서 factorial 4를 계산한 다음, 그 결과에 5를 곱해서 결과를 얻는다. 앞의 프로그램은 두 가지 큰 차이가 있다. 하나는 반복되는 계산을 루프를 통해 해결한 것과 함수 호출을 통해 해결한 것이다. 또 다른 하나는 변수에 저장되어 있는 값을 수정하면서 계산하는 것과 함수를 사용해서 계산 결과를 들고 다니면서 계산해 나가는 것이다.

때로는 수학적 함수를 기반으로 작성하는 것보다 메모리 반응을 기반으로 작성하는 것이 편리할 때가 있다. 그러한 경우를 위해서 nML은 기존의 방식대로 프로그래밍할 수 있도록 변경 가능한 메모리 장소와 배열 그리고 루프 명령어를 지원하고 있다.

변경 가능한 메모리 장소를 만들고 값을 수정하는 방법은 다음과 같다. ref를 이용해 장소를 만들고 '!'를 이용해 값을 읽은 후 ':='를 이용해 값을 변경한다.

```
# val x = ref 1;;
val x: int ref = {contents = 1}
# x := !x + 1;;
val it: unit = ()
# !x;;
val it: int = 2
```

초기값을 1로 하는 정수형 메모리 장소를 만든다. 여기서 x는 만들어진 메모리 장소의 주소를 말한다. 그 장소에 저장된 값을 읽어 1 증가시킨 값으로 변경한 뒤, 저장된 값을 읽어 보았다. 주의할 점은 nML에서는 메모리 장소의 주소와 저장된 값을 정확히 구분한다는 것이다. 'x := !x + 1'에서 x는 주소를 의미한다. 그렇기 때문에 오른쪽에서는 x의 값을 의미하므로 !x라고 해야 하고(!는 주소에서 읽어 온 값을 의미한다), 왼쪽에서는 그 주소에 저장하는 것이므로 x라고 해야 한다. 또한 nML에서는 주소의 연산을 금지한다. 주소의 연산은 C, C++에서 많은 오류를 양산하는데, 사실 주소의 연산은 배열 외에는 필요치 않다.

배열은 기존의 언어와 유사하게 사용된다. 배열을 표시하기 위해 괄호 [| |]를 사용한다.

```
# val x = [| 1, 2, 3, 4 |];;
val x: int array = [|1, 2, 3, 4|]
# val y = x.[1];;
val y: int = 2
# x.[2] <- y;;
val it: unit = ()
# x;;
val it: int array = [|1, 2, 2, 4|]
```

4개짜리 정수 배열을 만들고 그 중 두 번째(C, C++와 마찬가지로 첫 번째를 0번으로 한다) 것을 읽은 뒤, 세 번째를 읽은 값으로 변경하고 그 결과를 보았다. 다차원 배열은 기본적으로 지원하지 않지만 준비된 라이브러리를 사용하면 된다(http://ropas.kaist.ac.kr/n/lib).

루프를 구현하려면 while문과 for문을 사용하면 된다. while문의 모양새는 기존의 언어와 유사하다. 1부터 100까지 정수의 합을 구하는 프로그램을 while로 작성해 보자.

```
# val sum = ref 0 and i = ref 1;;
# while !i <= 100 do
    sum += !i;
    i++
  end;;
# !sum;;
val it: int = 5050
```

증가 장소 i와 누적 장소 sum을 준비하고, while문을 사용해 i를 1씩 증가시키면서 sum에 누적하는 것을 반복한다. 'sum += i'는 예측했겠지만 'sum := !sum + i'를 줄여 쓴 것이고, i++는 'i := !i + 1'을 줄여 쓴 것이다. 그리고 두 명령어 사이의 ';'은 차례대로 수행하라는 뜻이다. 결과는 알다시피 5050이다.

for문은 기존의 언어와 약간 다른 모습을 갖고 있다. C 언어에서는 for문이 다양한 형태로 사용될 수 있지만, nML에서는 for문의 기본적인 형태만을 허용한다. 앞의 while문으로 작성한 프로그램을 for문으로 작성하면 다음과 같다.

```
# val sum = ref 0;;
# for i=1; i<=100; i+1 do sum += i end;;
# !sum;;
val it: int = 5050
```

여기서 i는 while로 작성했을 때와는 달리 변경 가능한 장소가 아님을 주의하기 바란다. 결과적으로 이러한 기능을 사용해 factorial 함수를 다음과 같이 작성할 수도 있다.

```
fun factorial n =
  let val acc = ref 1
  in  for i=1; i<=n; i+1 do acc *= i end;
      !acc
  end
```

이렇게 C로 작성한 것과 유사하게 nML로 표현할 수 있다. 그러나 필자는 이렇게 프로그래밍하는 것을 nML 사용자에게 권장하지 않는다. nML에서는 일반적으로 nML 스타일로 작성해야 단순 명료해지기 때문이다. 물론, 메모리 반응을 기반으로 작성해야 단순 명료해질 때는 nML 스타일을 고집하면 안 된다. nML에서는 양쪽 다 지원하므로 선택은 사용자의 몫인 것이다.

다형 타입

자, 이제 본격적으로 코드 재사용을 위한 여러 기능에 대해 알아보자. C 프로그래밍 경험이 있는 사람은 비슷한 모양의 코드를 하나로 합치고 싶은 욕구를 많이 느꼈을 것이다. 그러나 C에서는 하나로 통합하는 것이 여간 쉽지 않다. 어려운 경우 중에 하나가 타입이 맞지 않는 경우다. 예를 들어, 정수형 리스트의 길이를 재는 코드와 문자열형 리스트의 길이를 재는 코드를 작성했다고 하자. 두 코드를 비교해 보면 타입과 상관없이 똑같은 일을 하고 있음을 알 수 있다. 그러나 C에서는 자연스럽게 타입이 다른 두 코드를 통합하는 방법을 제공하지 않는다.

nML에서는 하나의 코드가 여러 타입으로 사용되도록 할 수 있다. 우리는 이것을 다형 타입이라고 부른다. 앞서 이야기한 정수형 리스트의 길이를 재는 코드와 문자열형 리스트의 길이를 재는 코드를 한 번에 작성해 보자.

```
# fun length l =
    case l of
      [] => 0
    | h::t => 1 + length t;;
val length: 'a list -> int = <fun>;
```

함수의 타입이 흥미롭게 생겼다. 함수 타입 'a list -> int는 'a list 타입의 값을 받아 정수를 주는 함수를 뜻한다. 여기서 'a는 아무 타입이나 될 수 있다는 것이다. 정말 그런지 확인해 보자.

```
# length [1, 2, 3];;
val it: int = 3
# length ["a", "b"];;
val it: int = 2
```

length의 인자로 정수 리스트든 문자열 리스트든 상관없이 리스트의 길이를 결과로 준다. 또 다른 다형 타입 함수를 작성해 보자. 작성하려는 함수는 튜플을 받아 바꿔주는 함수다.

```
# fun swap (a,b) = (b,a);;
val swap: 'a * 'b -> 'b * 'a = <fun>
# swap (1, "a");;
val it: string * int = ("a", 1)
# swap ([1,2], 1.5);;
val it: real * int list = (1.5, [1, 2])
```

swap 함수의 타입을 보면 'a * 'b -> 'b * 'a이다. 튜플을 받아 튜플을 주는 함수인데, 입력 튜플과 결과 튜플이 서로 뒤바뀐 타입이다.

이렇게 nML이 제공하는 다형 타입은 간단하게 코드를 재사용하도록 해준다. 특별한 정의 없이 함수를 작성하면 알아서 타입을 유추해서 다형 타입 함수로 만들어 준다. 만들어진 다형 타입 함수는 매번 사용될 때마다 다른 타입으로 사용될 수 있다. 이러한 다형 타입 함수는 다음에 소개될 타입 인자를 받는 데이터 타입과 함수를 주고받는 방법과 혼용되어 보다 간단한 프로그래밍을 가능하게 한다.

타입 인자를 받는 데이터 타입

지난 기사에서 다룬 사용자 정의 타입을 통해 nML에서는 복잡한 데이터 구조를 간단하게 정의하고 사용할 수 있었다. 그런데 다형 타입에 대해 알고 나니 한 가지 욕심이 생긴다. 다형 데이터 구조를 만들 수는 없을까? 즉, 정수형 트리 구조와 문자열형 트리 구조를 한 번에 정의할 수 없을까? 인자를 받는 사용자 정의 타입을 사용하면 구조는 같고 내부에 쓰이는 타입이 다른 데이터 구조를 한 번에 정의할 수 있다. 지난 시간에 다룬 정수형 트리 구조의 정의는 다음과 유사했다.

```
type itree = ILeaf
           | INode of int * itree * itree
```

정수 대신 문자열을 갖는 트리구조를 정의하면 아마 다음과 같을 것이다.

```
type stree = SLeaf
           | SNode of string * stree * stree
```

두 트리 구조는 노드가 갖고 있는 값의 타입이 다르다는 것을 제외하고는 똑같은 구조를 지니고 있다. 두 트리 구조가 다른 부분을 매개변수로 두고 다음과 같은 트리 구조를 만들 수 있다.

```
type 'a tree = Leaf
             | Node of 'a * 'a tree * 'a tree
```

여기서 'a는 아직 어떤 타입이 될지 모르는 인자 부분이다. 'a를 여러 타입으로 바꿈으로써 여러 데이터 타입을 만들 수 있다. 예를 들어 정수형 트리 구조는 int tree 타입이고, ('a = int) 문자열형 트리 구조는 string tree 타입이 되는 것이다.

```
('a = string).
```

그럼, 트리를 만들어 보자.

```
# Node(1, Node(2, Leaf, Leaf), Leaf);;
val it: int tree = ...
# Node("a", Node("b", Leaf, Leaf), Leaf);;
val it: string tree = ...
```

다시 되짚어 보면 리스트가 바로 인자를 받는 데이터 타입이다. 정수형 리스트는 int list, 문자열형 리스트는 string list였던 것이다. 인자를 받는 데이터 타입에 대한 계산을 다형 타입 함수를 사용해 한 번에 작성할 수 있다. 예를 들어, 앞에 정의한 트리 구조에서 Leaf의 개수를 세는 함수를 작성해 보자.

```
# fun count t =
    case t of
      Node(_, t1, t2) => count t1 + count t2
    | Leaf => 1;;
val count: 'a tree -> int = <fun>
# count Node(1, Node(2, Leaf, Leaf), Leaf);;
val it: int = 3
# count Node("a", Leaf, Leaf);;
val it: int = 2
```

이런 식으로 여러 타입의 값에 대해 동일한 계산을 하는 함수를 한 번에 작성할 수 있다. 다형 타입 함수의 이러한 면모는 다음에 소개할 함수를 주고받는 방법으로 보다 유용하게 쓰일 수 있다.

함수를 주고받는 함수

코드를 재사용하고 싶은 경우에 걸림돌이 되는 것은 타입만은 아니다. 예를 들어 두 코드가 특정한 부분을 제외하고는 똑같을 때, 공통된 부분을 다시 작성하지 않고 재사용하고 싶을 때가 많다. 예를 들어, 리스트 {1,2,3}이 있을 때, 1씩 더한 리스트 {2,3,4}를 얻는 코드와 1씩 뺀 리스트 {0,1,2}를 얻는 코드는 더하고 빼는 것만 다를 뿐 다른 부분은 똑같을 것이다.

함수를 주고받는 방법(higer-order function)으로 재사용하면 자연스럽게 해결할 수 있다. nML에서는 함수를 특별하게 생각하지 않는다. 함수도 정수, 문자열과 같이 값일 뿐이다. 이것이 의미하는 것은 함수가 정수, 문자열과 같이 함수의 인자 또는 결과로 자연스럽게 사용된다는 것이다. 이렇게 함수를 인자 또는 결과로 사용하면 코드의 재사용을 극대화시킬 수 있다.

함수를 인자로 받는 경우를 다음 map 함수를 통해 보자. map 함수가 하는 일은 함수와 리스트를 받아서 리스트의 각각의 원소에 함수를 적용해 얻어지는 결과를 리스트해 준다.

```
# fun map f l =
    case l of
      [] => []
    | h::t => f h :: map f t;;
val map: ('a -> 'b) -> 'a list -> 'b list = <fun>
```

map 함수가 하는 일만큼 타입이 조금 복잡하게 생겼다. 차례로 읽어 보면 map 함수는 먼저 'a -> 'b 타입의 함수를 받고 'a list 타입의 리스트를 받아 'b list 타입의 결과 리스트를 준다. 기존의 언어에만 익숙한 독자들은 '이게 도대체 뭐하는 함수인가?' 라고 할 것이다. map 함수를 이해하기 위해 직접 사용해 보자.

```
# fun incr x = x + 1;;
val incr: int -> int = <fun>
# map incr [1, 2, 3];;
val it: int list = [2, 3, 4]
# fun decr x = x - 1;;
val decr: int -> int = <fun>
# mao decr [1, 2, 3];;
val it: int list = [0, 1, 2]
```

먼저 정수에 1을 더한 결과를 주는 incr 함수를 만든다. map의

첫 번째 인자로 incr 함수를 주고 두 번째 인자로 [1, 2, 3]을 주면 [2, 3, 4]를 준다. 반대로 1을 빼는 decr 함수를 주면 [0, 1, 2]를 준다. 이 예제를 보면 map 함수가 무엇을 하는 건지 감이 잡힐 것이다. 처음에 이야기한 리스트의 원소를 1씩 증가/감소시키는 코드에서 공통된 부분을 map 함수로 작성한 것이다. map 함수는 각 원소에 적용할 함수를 인자로 받기 때문에(게다가 다형 타입이기 때문에) 다양하게 사용할 수 있다.

```
# String.length;;
val it: string -> int = <fun>
# map String.length ["one", "two", "three"];;
val it: int list = [3, 3, 5]
```

여기서 String.length는 문자열의 길이를 주는 nML에 내장된 함수다. 주목할 점은 String.length는 인자와 결과의 타입이 다른데도 사용할 수 있다는 것이다. 다시 되짚어 보면 map의 첫 번째 인자 타입은 'a -> 'b였다. 인자와 결과의 타입이 같을 필요는 없다. 그러나 그 다음 인자인 리스트는 'a list 타입이므로 반드시 첫 번째 인자 함수의 인자 타입과 맞아야 한다. 즉, 앞 예제에서 'map String.length [1,2]' 라고 하면 오류가 발생한다.

이외에도 많이 쓰이는 중요한 함수를 주고받는 함수들은 많이 있다. 그 중에서도 자주 쓰이는 함수들은 nML에서 기본적으로 내장하고 있다(http://ropas.kaist.ac.kr/n/lib). 리스트 구조에 대해 자주 쓰이는 함수들은 List 모듈로, 배열에 대해 자주 쓰이는 함수들은 Array 모듈로 제공하고 있다. 다음은 모듈에 대해서 알아보기로 하자.

모듈

nML에서 모듈은 단지 여러 정의들을 모아 놓은 것일 뿐이다. 이미 많은 독자들이 모듈이라는 단어를 여기저기서 접해 보았을 것이다. 그리고 모듈에 특별한 의미를 두는 언어들도 접해 보았을 것이다. nML에서 모듈은 특별한 의미 없이 단지 타입 정의, 값 정의, 예외상황 정의, 모듈 정의들을 모아 놓은 것이다.

정수형 스택을 위한 모듈을 구현해 보자. 정수형 스택을 정수 리스트로 구현하고 empty, push, pop 등을 구현하면 될 것이다. 스택이 비었을 때 pop하면 예외상황 Empty를 발생시킨다고 하자. 이러한 모듈을 다음과 같이 구현할 수 있다.

```
# structure Stack =
    struct
      type t = int list
      exception Empty
      val empty = []
      fun push x t = x :: t
      fun pop t =
        case t of
          [] => raise Empty
        | h::t => (h, t)
    end;;
```

모듈은 structure로 이름 짓고 'struct … end' 내에 정의들을 기술한다. nML 대화형 컴파일러(nml)에서 이와 같이 작성하면 결과 메시지가 나올 것이다. 결과 메시지가 기술하고 있는 것은 모듈의 타입이다. 즉, 모듈 내에 어떤 것들이 있는지 타입 수준에서 기술해 준다. 이렇게 모듈을 정의하면 모듈의 타입을 컴파일러가 알아서 유추해 준다.

모듈 타입은 기본적으로 유추해 주지만 사용자가 직접 기술해 주면 여러 가지 효과를 볼 수가 있다. 답부터 이야기하면 정보를 가릴 수 있다(information hiding). 우선 모듈 타입 STACK을 정의해 보자.

```
# signature STACK =
    sig
      type t
      exception Empty
      val empty : t
      val push : int -> t -> t
      val pop : t -> int * t
    end;;
```

모듈 타입은 signature로 이름 짓고 'sig … end' 내에 명세를 기술한다. 앞의 모듈 타입 명세는 쉽게 무슨 뜻인지 짐작되리라 믿는다. 주목할 점은 type t가 어떤 타입이라는 것을 일부러 기술하지 않았다는 것이다. Stack 모듈을 STACK 모듈 타입에 맞추어서 새로운 모듈 Stack'을 만들어 보자.

```
# structure Stack' = Stack : STACK;;
```

조금 전 Stack 모듈을 정의했을 때와는 결과 메시지가 type t 부분에서 다를 것이다. 두 모듈 Stack과 Stack'이 어떻게 다른지 직접 사용해 보자.

```
# Stack.push 1 [];;
val it: int list = [1]
# Stack'.push 1 [];;

오류: Stack'.t 타입이어야 하는데 _a list 타입입니다.
```

똑같이 빈 스택([])에다가 1을 집어넣었는데, Stack 모듈을 사용할 때는 문제가 없는 반면 Stack' 모듈을 사용할 때는 오류가 난다. Stack' 모듈에서는 스택의 타입이 int list인지 모르기 때문에 빈 리스트([])가 스택이라고 인정하지 않는 것이다. 실제로 리스트가 아닌 다른 방법으로 스택을 구현하면 빈 스택이 []이 아닌 다른 것이 될 것이다. 그러면, 모듈 Stack'에서 빈 스택은 무엇인가? 바로 empty라는 값이다.

```
# Stack'.push 1 Stack'.empty;;
val it: Stack'.t =
```

구현의 의존성을 없애기 위해 정보를 가린 것이다(information hiding). 우리가 스택을 사용할 때는 스택이 어떻게 구현됐는지는 중요하지도 않고, 어떻게 구현됐느냐에 따라 우리가 사용하는 것이 달라진다면 불편하기만 할 뿐이다. Stack' 모듈은 스택이 어떻게 구현됐는지 숨기고, 단지 empty, push, pop, Empty만 제공해 사용자가 제공되는 것들로만 구현하도록 한 것이다. 타입 정보뿐만 아니라 값의 이름도 가릴 수 있다. 다음은 매 번 호출 때마다 1씩 증가시킨 값을 주는 카운터(counter) 모듈이다.

```
# structure Counter =
    struct
      val x = ref 0
      fun counter () = x++; !x
    end :
    sig
      val counter : unit -> int
    end;;
structure Counter: sig val counter: unit -> int end
```

모듈에서는 x와 counter 두 가지를 정의했는데, 뒤의 모듈 타입(sig … end)에서는 counter만 포함하고 있다. 직접 사용해 보자.

```
# open Counter;;
# counter();;
val it: int = 1
# counter();;
val it: int = 2
# Counter.x := 10;;
오류 : Counter.x을(를) 모릅니다.
```

첫 줄의 open은 모듈 사용을 번거롭지 않게 하기 위해 여는 역할을 한다. 원래는 Counter.counter()라고 호출해야 하는데 앞머리인 Counter.를 생략해도 되는 것이다. 카운터 기능을 잘 수행하고 있다. 마지막에 카운터를 변경시키려고 했는데, Counter.x를 모른다고 오류 메시지가 난다. Counter.x는 가려졌기 때문에 counter 함수를 통하지 않고는 값을 변경시킬 수가 없는 것이다.

이렇게 모듈로 단순히 모아두고 모듈 타입으로 정보를 가림으로써 모듈별로 프로그래밍할 수 있다. 다음에는 이미 작성된 모듈 템플릿으로 새로운 모듈을 생성하는 방법, 모듈 함수를 소개하도록 한다.

모듈 함수

모듈 함수(functor)는 말 그대로 모듈을 받아서 모듈을 주는 함수다. 기본적으로 코드 재사용을 위해서 비슷한 내용의 모듈들을 일반적으로 한 번에 작성하는 데 쓰인다. 예를 들어, 집합(set) 모듈을 작성한다고 하자. 우리가 필요한 것은 원소가 정수인 집합 모듈과 원소가 문자열인 집합 모듈이라고 하자. 여러 방법이 있겠지만 모듈 함수를 사용해 구현할 수 있다. 우선 타입과 그 타입의 비교 연산자를 제공하면 그 타입을 원소로 갖는 집합 모듈을 주는 모듈 함수를 구현한다. 그러면 구현된 모듈 함수를 이용해 쉽게 정수형 집합 모듈, 문자열형 집합 모듈들을 생성해 낼 수 있다. 모듈 함수의 또 다른 기능은, 각각의 모듈에 입력 모듈 타입과 결과 모듈 타입을 정확히 기술함으로써 모듈간 구현 의존성을 분리하는 역할도 할 수 있다.

집합(set)을 구현해 보면서 모듈 함수를 이해해 보자. 구현하고자 하는 입력 모듈 타입 EQ와 결과 모듈 타입 SET은 다음과 같다.

```
signature EQ =
  sig
    type t
    val equal : t -> t -> bool
  end
signature SET =
  sig
    type element
    type t
    val empty : t
    val member : t -> element -> bool
    val add : t -> element -> t
  end
```

집합을 구현하기 위해 필요한 것은 원소의 타입이 무엇인가와 원소들끼리 비교하는 방법을 알아야 한다. 최소한 두 원소가 같은지를 결정할 수 있어야 한다. 그래서 입력 모듈 EQ에는 원소의 타입 t와 t 타입의 값들이 같은지 비교하는 함수 equal을 넣었다. 결과 모듈은 간단하게 공집합을 구현한 empty, 원소가 집합에 속

하는지 알아보는 member, 집합에 원소를 넣는 add 만을 구현하기로 하자.

functor 구문을 사용해 모듈 함수를 정의한다. 집합 모듈을 위한 모듈 함수는 다음과 같이 구현할 수 있다.

```
functor SetFn( X : EQ ) : SET where type element = X.t =
  struct
    type element = X.t
    type t = element list
    val empty = []
    fun member s x =
      case s of
        [] => false
      | h::t => X.equal h x || member t x
    fun add s x = if member s x then s else x :: s
 end
```

모듈 함수 이름은 SetFn이고 입력은 EQ 타입의 모듈 X이고 결과는 SET 타입인데, 단 결과 모듈의 element 타입이 입력 모듈의 t 타입과 같다는 뜻이다. 모듈 함수 내부를 보면 입력으로 주어진 X.t, X.equal이 사용된 것을 볼 수 있다.

원소(element)는 입력 모듈의 t 타입(X.t)과 같고, 집합은 원소의 리스트로 구현했다. 공집합은 []이고 member, add 함수들은 X.equal을 사용해 작성했다. 참고로 ||는 논리합(or) 연산자이다. 모듈 함수에 모듈을 입력으로 주면 새로운 모듈이 생성된다. 정수형 모듈을 정의하고 구현된 모듈 함수 SetFn에 적용해 보자.

```
# structure IntTy =
    struct
      type t = int
      fun equal x y = (x = y)
    end;;
# structure IntSet = SetFn(IntTy);;
# open IntSet;;
# val x = add (add empty 5) 6;;
# member x 5;;
val it: bool = true
# member x 1;;
val it: bool = false
```

IntTy 모듈에 타입 t가 정수형이고 정수 두 개가 같은지를 알려주는 equal 함수를 정의하였다. 두 번째 # 줄을 보면 IntTy 모듈을 모듈 함수 SetFn에 주고 얻어진 모듈을 IntSet이라고 이름지었다. IntSet 모듈은 정수형 집합 기능을 잘 수행하고 있다. 공집합에 5와 6을 넣은 집합을 x라 하고, 5와 1이 x에 속하는지 검사해 보았다. 비교 함수가 일반적이지 않은 다른 집합을 만들어 보자. 원소는 정수 두 개이고 (a, b)와 (b, a)는 같은 원소로 봤을 때 다음과 같이 구현할 수 있다.

```
# structure IntPairTy =
    struct
      type t = int * int
      fun equal (x1,y1) (x2,y2) =
        x1 = x2 && y1 = y2 || x1 = y2 && y1 = x2
    end;;
# structure IntPairSet = SetFn(IntPairTy);;
# open IntPairSet;;
# val x = add (add empty (1,2)) (3,4);;
# member x (4,3);;
val it: bool = true
# member x (1,3);;
val it: bool = false
```

equal의 구현에서 눈치챘겠지만 &&는 논리곱(and) 연산자이다. 결과를 보면 집합 x는 (1, 2)와 (3, 4)를 원소로 갖고 있는데, 의도대로 (4, 3)도 x에 속한다고 결정한다.

모듈과 모듈 함수를 이용하면 여기 예시한 간단한 기능뿐만 아니라 프로그램을 모듈별로 나눠서 작성하는 데 도움이 된다. 즉, 큰 프로그램을 작성하기 위해서는 모듈과 모듈 함수가 필수적이다. 그러면 여기서 이런 질문이 떠오른다. 큰 프로그램을 작성하려면 파일을 여러 개로 나눠서 작업해야 할 텐데, nML에서는 어떻게 해야 할까? 답은 간단하다. 유닉스에서 C 프로그래밍하듯 하면 된다.

여러 파일로 나눠서 프로그래밍하기

여러 파일로 나눠서 프로그래밍하는 방법은 간단하다. 파일이 a.n, b.n으로 나눠져 있고 b.n에서만 a.n의 것을 사용한다면 차례로 컴파일하고 나서 링크해 주면 된다. 예를 들어, 다음 두 프로그램을 보자.

```
structure A =
  struct
    val x = 10
  end
```

프로그램 1: a.n

```
val _ = Format.printf "%d\n" A.x
```

프로그램 2: b.n

a.n에서 x를 10으로 정의하고 b.n에서 그 값을 읽어 화면에 출력했다. 주의할 점은 b.n에서 a.n의 이름을 사용하기 위해서는 이름이 어떤 모듈 내에 정의돼 있어야 한다. 예를 들어, a.n이 'val x=10' 라고만 되어 있으면 nML 컴파일러가 x가 어디에 정의된 이름인지 찾아 내지 못한다.

또 하나 주의할 것은 파일 내에는 대화형 컴파일러 때와는 달리 모두 정의로 이뤄져 있어야 한다. b.n을 보면 명령문을 사용하기 위해 'val _ =' 를 덧붙여 정의로 바꾸어 사용했다. 마지막으로 ';;' 는 파일 내에 사용하면 안 된다. ';;' 는 대화형 컴파일러에서 입력을 끝내는 특수한 신호일 뿐이다. 앞의 두 프로그램으로부터 다음과 같이 실행 파일을 얻을 수 있다.

```
c:\> nmlc -c a.n
c:\> nmlc -c b.n
c:\> nmlc a.cmo b.cmo -o run.exe
c:\> run
10
```

첫 번째, 두 번째 줄은 .n 파일을 컴파일해 .cmo 오브젝트(object) 파일을 생성해 준다. 여기서 -c 옵션은 컴파일만 하라는 옵션이다. 세 번째 줄은 컴파일된 오브젝트 파일들을 링크하라는 명령어다. -o run.exe 옵션은 만들어지는 실행 파일 이름을 run.exe으로 하라는 것이다. 유닉스 환경에서는 .exe를 붙이지 않아도 좋다.

파일의 수가 많아지면 이것도 번거롭기 때문에 nmakegen을 사용한다. 유닉스에서 C로 프로그래밍할 때는 Makefile을 사용해 이런 번거로움을 피했는데 nml도 마찬가지로 Makefile을 사용한다. 그런데 Makefile 파일 만들기도 여간 귀찮은 것이 아니다. 그래서 제공하는 것이 nmakegen이다. nmakegen은 현재 디렉토리 내에 존재하는 .n 파일을 모두 찾아서 서로 의존 관계를 파악한 뒤, 차례대로 컴파일하고 링크해 주는 Makefile을 만들어 준다.

```
c:\> nmakegen
c:\> make
c:\> run
10
```

여기서 make는 유닉스에서 기본적으로 Makefile을 처리하는 실행 파일 이름이다. nML의 윈도우용 배포판에는 GNU make 실행 파일이 포함되어 있다. 여기서 run은 nmakegen이 기본적으로 만들어 주는 실행 파일 이름이다. nmakegen 덕분에 번거로운 일이 많이 줄어들었다.

궁금한 사항은 ropas 홈페이지로

2회에 걸쳐 nML이 어떤 언어인지 개략적이나마 기능을 나열해 보았다. 어떤 독자들은 다행히 대략적이나마 파악했을 수도 있고, 어떤 독자들은 좀 신기하긴 한데 무엇에 쓰는 물건인지 모른다고 할 수도 있다. 처음 소개하는 것이라서 자세한 내용보다는 어떤 기능들이 있는지 간단한 예제를 통해 나열하는 수준이어서 독자들의 이해에 한계가 있으리라 짐작한다.

좀더 관심 있는 독자들을 위해서 현재 문서화되어 있는 것들을 소개하면서 이번 호를 마치고자 한다. 아직은 nML이 개발 단계에 있어서 독자들의 욕구를 모두 채워 줄 자세한 한글 문서를 마련하지는 못했다. 조금이나마 마련된 것들은 http://ropas.kaist.ac.kr/n에 있고 앞으로 나올 것들도 모두 여기에 게시할 것이다. 궁금한 점은 게시판에 글을 올리거나 n@ropas.kaist.ac.kr로 물어봐도 좋을 것이다. 영문에 거부감이 없는 독자라면 nML의 친구들인 미국에서 개발된 SML/NJ(Standard ML of New Jersey), 프랑스에서 개발된 OCaml(Objective Caml)용 입문서를 참고 자료에 소개했으니 읽어 보기 바란다.

정리 : 강경수 elegy@sbmedia.co.kr

참고 자료

❶ The nML Programming Language System, Research On Program Analysis System, KAIST, http://ropas.kaist.ac.kr/n
❷ The Standard ML of New Jersey, Lucent Technologies, http://cm.bell-labs.com/cm/cs/what/smlnj
❸ The Object Caml Language System, Institut National de Recherche en Informatique et en Automatique, France, http://caml.inria.fr
❹ Introduction to Programming Using SML, Michael R. Hansen and Hans Rischel, Addison-Wesley, 1999.
❺ The Little MLer, Matthias Felleisen and Daniel P. Friedman, MIT Press, 1998
❻ The Functional Approach to Programming, G. Cousineau and M. Mauny, Cambridge University Press, 1998.
❼ Elements of ML Programming, Jeffrey D. Ullman, MIT Press, 1997.
❽ ML for the Working Programmer, Lawrence C. Paulson, Cambridge University Press, 1996.
❾ Abstract Data Types in Standard ML, Rachel Harrison, John Wiley & Sons, 1993.