nML과 함께 하는 프로그래밍 여행 3

nML 프로그래밍의 쓰임새 I

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 세 번에 걸쳐 본격적으로 nML이라는 언어를 소개하고자 한다. nML 프로그래밍 환경을 설치하는 방법에서부터 시작해 가벼운 프로그래밍까지 nML을 체험해 볼 수 있을 것이다.

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 : nML 프로그래밍의 쓰임새 II
5회 : nML 언더그라운드 콘서트

지난 호까지 연재를 통해 독자들이 과연 nML이 어떤 언어인가에 대한 궁금함과 기대감을 조금이나마 갖게 됐으리라 생각한다. 이번 호에 이르러서야 nML이라는 언어를 본격적으로 소개하게 됐는데, 아무래도 첫 번째 시간이기에 독특하고 멋진 내용보다는 기본적인 내용을 많이 다루고자 한다.

필자의 목적은 독자들이 이 글을 참조해 nML 프로그래밍을 할 수 있도록 하겠다는 것이 아니라, nML을 사용해 볼 생각이 들도록 만들려는 것이므로 너무 구체적인 설명은 가급적 생략하겠다. 필자는 nML의 장점을 부각시키기 위해 종종 C, C++, 자바 등의 프로그래밍 언어와 비교하는 방법을 사용했다. 글 중에서 특별히 지칭하지 않고 '다른 프로그래밍 언어'라고 한 경우는 대부분 C/C++/자바를 염두에 두고 읽으면 무리가 없을 것이다.

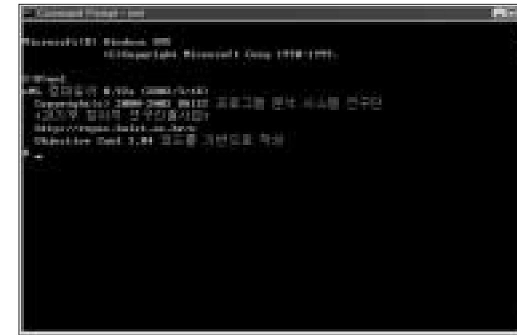
프로그래밍 환경 만들기

무엇보다도 먼저 nML을 사용할 수 있는 환경을 만드는 작업이 필요하다. nML 컴파일러는 nML 홈페이지(http://ropas.kaist.ac.kr/n/)에서 다운로드해 설치할 수 있다. nML 컴파일러는 유닉스 환경과 윈도우 환경 모두 설치해 사용할 수 있으나, 여기서는 윈도우 환경에 설치하는 경우를 예로 설명하겠다.

nML 컴파일러의 설치는 어렵지 않다. 먼저 최신 버전의 윈도우용 컴파일러(http://ropas.kaist.ac.kr/n/nml-0.92a.zip)를 다운로드해 적당한 폴더에 압축을 푼다. 그리고 나서 다음과 같이 두 개의 환경 변수를 설정해 주면 설치가 끝난다. 만약 C:\nml이라는 폴더에 압축을 풀었다면 NMLLIB라는 환경 변수를 만들어 C:\nml\lib로 설정하고, PATH에 C:\nml\bin과 C:\nml\lib를 추가하면 된다. 시스템 등록 정보의 고급 탭에서(윈도우 2000 기준으로) 환경 변수와 관련된 기능을 쉽게 발견할 수 있을 것이다.

이상의 과정을 마쳤다면 nML 프로그래밍 환경이 준비됐는지 확인해 보자. 유감스럽게도 아직 쓸만한 통합 환경이 제공되지 않으므로, 대개의 경우 커맨드 프롬프트를 통해 nML 환경을 사용하게 된다. 커맨드 프롬프트를 띄우고 nml이라고 입력하면 〈화면 1〉과 같은 메시지를 볼 수 있다.

〈화면 1〉 nML의 실행 화면



이것은 nML의 대화식 프로그래밍 환경인 nml.exe를 실행시킨 결과다. # 기호는 nML 대화식 환경의 프롬프트이다. 이 시점에서는 그냥 '#quit;;'를 입력해 대화식 환경을 빠져 나오도록 하자.

대화식 환경은 프로그램을 부분적으로 수행하고 중간 결과와 내부 정보, 타입 등을 쉽게 확인할 수 있도록 해 준다. 다시 말해 대화식 환경에서 어떤 프로그램을 한 줄 씩 입력하면, 최종적으로는 그 프로그램을 컴파일해 실행한 것과 동일한 결과를 얻겠지만, 필요에 따라 그 중간 과정의 계산 값을 확인해 볼 수도 있다는 말이다. 대화식 환경을 사용하면 완전한 프로그램을 만들지 않고도, nML의 각 구문이 어떻게 동작하는지를 쉽게 확인해 볼 수 있으므로 nML을 처음 익힐 때 매우 유용하다. 물론 nML에 익숙해진 이후에도 대화식 환경은 테스팅과 디버깅 작업에 매우 유용하게 사용된다.

앞으로 많은 부분을 대화식 환경을 이용하면서 이야기를 진행하겠지만, 그 전에 nML 컴파일러를 사용해 간단한 실행 파일을 만들어 보도록 하자.

〈리스트 1〉 hello.n

```
1:      fun main () =
2:        let
3:          val s = "Hello, World!\n"
4:        in
5:          print_string s
6:        end
7:
8:      val _ = main ()
```

Hello, World!

Hello, World! 프로그램은 거의 모든 프로그래밍 언어의 입문서에 등장하는 전통적인 예제 프로그램이다. 전통을 살려 이 글에서도 nML 컴파일러의 동작을 체험해 보는 데, 이 예제를 사용하려고 한다. 조금 지겹더라도 독자 여러분들의 양해를 바란다.

먼저 〈리스트 1〉을 입력해 적당한 폴더에 저장하자. 필자는 C:\nml\src 폴더를 만들어 사용했다. 각자 선호하는 에디터를 사용해 프로그램을 작성하면 된다. 전형적인 Hello, World! 프로그램보다는 조금 복잡하다고 느낄 것이다. 사실 Hello, World!라는 문자열을 출력하는 것만이 목적이라면 이렇게 복잡하게 만들 필요는 없지만, 많은 면을 보여주기 위해 일부러 조금 복잡하게 만들었다. 또한 〈리스트 1〉은 의도적으로 〈리스트 2〉의 C 언어 프로그램과 비슷하게 보이도록 만들었으니, 찬찬히 〈리스트 2〉와 비교해 보면 각 부분의 의미를 대략 짐작할 수 있으리라 생각한다.

Hello, World!여기까지 아무런 문제가 없었다면, 이제 이 프로그램의 각 부분을 조목조목 파악해 보자. fun, val, let, in, end는

〈리스트 2〉 hello.c

```
1:      #include <stdio.h>
2:
3:      void main()
4:      {
5:        char s[] = "Hello, World!\n";
6:
7:        printf("
8:      }

이제 프로그램을 컴파일하고 실행해보자. 컴파일러 실행 파일의 이름은 nmlc.exe이다.

C:\NML\SRC>nmlc hello.n -o hello.exe

C:\NML\SRC>hello
```

모두 nML의 예약어다. fun은 함수를 정의할 때 사용하고, val은 값을 선언할 때 사용한다. 엄밀하게 말하면 nML에서는 함수도 값의 일종이지만, 이 시점에서는 일단 이렇게 알아두어도 무리가 없다.

값과 변수의 차이에 유의해야 한다. val로 선언한 s는 변수가 아니므로 선언의 유효 범위 내에서 다른 정수 값으로 바꾸는 것이 불가능하다. 변수 대신 값을 사용하여 프로그래밍 하는 것의 장점에 대해서는 이 연재의 첫 번째 기사에서 비교적 상세하게 언급했으므로 여기서는 생략하겠다.

let-in-end는 값의 유효범위(scope)를 제한하는 데 사용한다. C/C++/자바 프로그램에서는 블럭 {}에 이러한 의미가 포함되어 있다. 다음 프로그램과 〈리스트 1〉을 비교해 보자.

```
1:      val s = "Hello, World!\n"
2:
3:      fun main () = print_string s
4:
5:      val _ = main ()
```

print_string은 하나의 문자열 인자를 받아 인자로 주어진 문자열을 화면에 출력하는 함수다. 실제로는 Pervasives라는 모듈에 정의되어 있는데, 이 모듈은 nML에서 기본적으로 사용할 수 있는 타입, 함수 등을 포함하고 있다. Pervasives 모듈에 무엇이 정의되어 있는지를 살펴보려면 http://ropas.kaist.ac.kr/n/lib92/Pervasives.html을 참조하면 된다.

〈리스트 1〉에 오해를 살 만한 부분이 한 군데 있다. 〈리스트 2〉의 C 언어 프로그램과 비슷하게 보이도록, 일부러 main이라는 함수 이름을 사용했지만, 프로그램에 반드시 main이라는 이름의 함수가 있어야 하는 것은 아니다. 다시 말하면 프로그램의 수행이 시작되는 위치가 따로 정의되어 있는 것이 아니라, 그저 프로그램의 맨 위부터 차례대로 수행된다. 실제로 'Hello, World!'가 출력되는 이유는 마지막 줄에서 main 함수를 호출하기 때문이다.

마지막 줄의 의미가 조금 난해하다. 예약어 val은 조금 더 정확하게 말하면 값에 새로운 이름을 부여하는 것이다. 여기서는 값의 이름 부분을(밑줄을 사용해) 비워 두었다. 이렇게 하면 등호 오른쪽의 식을 계산하되 이름을 주지 않겠다는 것이므로, 단순히 등호 오른쪽의 식을 계산하는 셈이 된다. 이러한 방법은 자주 사용하게 되므로 그냥 외워두는 것이 좋다.

기본 타입

nML은 값 중심의 프로그래밍 언어다. nML 프로그램에서 기본적으로 사용할 수 있는 값에는 어떤 것들이 있는지 살펴보자.

여기서부터는 주로 대화식 환경을 이용해 진행할 것이다. 앞에서 말했듯이 대화식 환경을 이용하면, 완전한 프로그램을 만들지 않고도 수행 결과를 확인할 수 있어 편리하기 때문이다. 대화식 환경을 실행시키고 다음과 같이 입력해 보자. 입력의 의미는 '1+1을 계산한 값을 앞으로 v라는 이름으로 사용하겠다'는 것이다. 연속되는 세미콜론 두 개는 대화식 환경에게 입력의 끝을 알리는 것이니 기억해 두자.

```
nML 컴파일러 0.92a (2002/3/18)
  Copyright(c) 2000-2002 KAIST 프로그램 분석 시스템 연구단
  (과기부 창의적 연구진흥사업)
  http://ropas.kaist.ac.kr/n
  Objective Caml 3.04 코드를 기반으로 작성
# val v = 1+1 ;;
val v: int = 2
#
```

대화식 환경의 출력을 살펴보자. 등호 오른쪽의 2는 물론 계산 결과이다. 콜론 오른쪽의 int는 예상했듯이 값 v가 nML 프로그램에서 가지게 되는 타입의 이름이고, 정수 타입을 의미한다.

앞에서 값 v의 타입은 자동으로 유추됐음을 볼 수 있다. 〈리스트 1〉을 돌이켜 봐도 프로그래머가 타입을 지정해 준 곳은 한 군데도 없었다. 예제들이 너무 간단해서 실감이 잘 나지 않겠지만, 매우 복잡한 프로그램인 경우에도 nML 컴파일러는 프로그래머의 도움 없이 프로그램 내 모든 부분의 타입을 정확하게 파악할 수 있다.

자동으로 타입을 유추해 줄 뿐만 아니라, nML 컴파일러를 통과한 프로그램은 실행 중에 타입 오류로 인해 문제가 발생하지 않는다는 것이 보장된다. 다시 말해 nML은 지난 호에서 언급한 대로 제 2세대 버그 잡는 기술을 충실하게 구현한 언어다. 이것은 nML의 가장 큰 장점 중 하나다.

그리고 이것은 프로그래머가 지정하지 않은 타입도 자동으로 유추할 수 있다는 의미이지, 프로그래머가 타입을 지정해 줄 수 없다는 의미가 아니다. 프로그래머가 원한다면 모든 타입 정보를 명확하게 지정해 주면서 프로그램을 작성할 수도 있다. 실제로 규모가 큰 프로그램을 작성할 때는, 부분적으로 타입을 지정해 놓으면 이후에 프로그램을 이해하는 데 또는 프로그램을 디버깅하는 데 도움이 된다.

몇 가지를 더 입력해 보면서 nML에는 어떤 기본 타입이 있는지를 확인해 보자. 각 타입에 대한 구체적인 설명은 필요하지 않으리라 본다.

```
# val 원주율 = 3.141592 ;;
val 원주율: real = 3.141592
# val 문자 = 'a' ;;
val 문자: char = 'a'
# val 문자열 = "안녕하세요.\n" ;;
val 문자열: string = "안녕하세요.\n"
# val 결과 = print_string 문자열 ;;
안녕하세요.
val 결과: unit = ()
```

마지막의 unit 타입에 대해서는 조금 설명이 필요할 것 같다. 인자가 필요하지 않은 함수를 unit 타입의 인자를 받는 함수로 정의하고, 아무런 값도 계산해 내지 않는 함수를 unit 타입의 값을 계산해 내는 함수로 정의한다. 대략 C/C++/자바의 void 예약어와 뜻이 통한다고 생각하면 되지만, 세부적인 면에서 조금 차이가 있다. 설명한 대로 〈리스트 1〉에서 main 함수는 인자를 받지 않는 함수가 아니라, unit 타입의 인자를 받는 함수다. '()'로 나타낸 것이 인자 부분을 비운 것이 아니라, '()' 자체가 unit 타입의 값이라는 사실이 재미있다.

이미 눈치 챘겠지만, 이 예제는 nML의 장점 한 가지를 더 소개하려는 의도를 갖고 만들었다. 그것은 값이나 함수의 이름에 한글을 자유롭게 쓸 수 있다는 점이다. 우리 나라에서 개발된 언어이니 마땅히 이래야 하지 않겠는가?

함수의 사용

이번에는 함수를 하나 만들어 보자. 그래도 무엇인가 쓸모있는 함수를 만드는 것이 좋겠으니, Euclid의 알고리즘을 사용해 두 수의 최대공약수를 계산하는 함수를 정의해 보자.

```
# fun gcd m n = if m = 0 then n else gcd (n ;
val gcd: int -> int -> int = <fun>
# gcd 9 12 ;;
val it: int = 3
```

우리가 정의한 함수의 타입이 흥미롭다. 타입 표현에서 화살표(➡)는 함수를 의미하는데, 'int ➡ int'는 정수를 인자로 받아 정수를 계산해 내는 함수들의 타입을 말한다. 그러면 'int ➡ int ➡ int'는 어떻게 읽어야 할까? 우선순위로 인해 이것은 'int ➡ (int ➡ int)'로 읽히는데, 이것은 정수를 받아 함수를 계산해 내는 함수를 의미하고 있다. 정말 그런지 확인해 보자.

```
# val gcd_9 = gcd 9
val gcd_9: int ➡ int = <fun>
# val gcd_9 12
val it: int = 3
```

이런 관점에서 보면 앞의 'gcd 9 12'는 실제로 '(gcd 9) 12'로 해석되어 수행된 셈이다. 기억하는 독자도 있겠지만 이것은 본지 2002년 4월호 '아름다운 언어 Haskell 프로그래밍 2'에서 김재우 씨가 설명한 커링(Currying)의 예다. Haskell과 마찬가지로 nML도 함수를 잘 지원하는 언어이고, Haskell이 지원하는 여러 가지 편리한 개념들이 nML에도 잘 구현되어 있다. 반드시 인자 두 개가 필요한 함수를 만들고 싶다면 어떻게 하면 될까? 아쉬운 대로 다음과 같이 할 수 있겠다.

```
# fun gcd (m,n) = if m = 0 then n else gcd (n ;
val gcd: int * int ➡ int = <fun>
# gcd (9,12);;
```

그럴듯해 보이지만, 엄밀히 말해 이것은 정수 인자 두 개를 받는 함수가 아니라, 튜플 타입의 인자 하나를 받는 함수를 정의한 것이다. 그러고 보니 튜플 타입에 대해 설명하지 않았는데, 이제 여러분에게 보다 복잡한 값인 튜플, 레코드, 리스트를 소개할 때가 되었다.

튜플, 레코드, 리스트

튜플, 레코드, 리스트 역시 nML에서 기본적으로 사용할 수 있는 값에 속한다. 여러 개의 기본 값들을 하나로 묶음으로써 만들어지는 새로운 값으로서, '정수와 실수의 튜플', '문자열의 리스트'와 같은 형태의 값을 말한다. 이처럼 비교적 복잡한 형태의 값을 별다른 사전 작업 없이 사용할 수 있다는 것은 매우 편리한 일이다. 간단하게 튜플, 레코드, 리스트를 사용해 보자.

```
# val tuple1 = (1, 2.0, "3") ;;
val tuple1: int * real * string = (1, 2, "3")
# val tuple2 = ((1, 2.0), "3") ;;
val tuple2: (int * real) * string = ((1, 2), "3")
# tuple1.0 ;;
val it: int = 1
# tuple2.0 ;;
val it: int * real = (1, 2)
```

이 예제에서는 두 개의 튜플 값을 선언하고 사용하는 예를 보였다. tuple1과 tuple2의 타입이 다르다는 것에 유의해야 한다.

tuple1의 첫 번째 값을 꺼낼 때는 'tuple1.0'과 같이 한다. 이

예제에서 'tuple1.3'은 정의되지 않는데, 만일 프로그램서 tuple1과 같은 값을 지정하고 'tuple1.3'을 사용하려고 한다면, 컴파일러가 오류를 지적해 줄 테니 걱정하지 않아도 된다.

```
# val record = { x = 0, y = 0, name = "원점" } ;;
val record: {name: string, x: int, y: int} = ("원점", 0, 0)
# record.x ;;
val it: int = 0;
# record.name ;;
val it: string = "원점";
```

이 예제에서는 레코드 값을 선언하고 사용하는 예를 보였다. 레코드와 튜플의 차이점은 레코드는 이름으로써 항목을 가리키고, 튜플은 순서로써 항목을 가리킨다는 점이다. 그리고 앞 예제에서 드러나듯이 레코드에서 항목의 순서는 별 의미가 없다.

레코드는 C/C++의 구조체(structure)와 비슷하다. 그러나 C/C++의 경우 사용하고자 하는 구조의 타입을 미리 선언해야 하지만, nML의 경우 그러한 번거로운 작업이 필요하지 않다.

```
# val list1 = [1, 2, 3] ;;
val list1: int list = [1, 2, 3]
# val list2 = 4::5::[] ;;
val list2: int list = [4, 5]
# List.nth list1 0 ;;
val it: int = 1
```

이 예제에서는 두 개의 리스트를 선언하고 사용하는 예를 보였다. 리스트와 튜플의 차이점 중 하나는 튜플은 서로 다른 타입의 값을 묶을 수 있지만 리스트는 같은 타입의 값만을 묶을 수 있다는 점이다.

list1과 list2를 선언할 때 리스트를 표현한 방법이 서로 다른데, 실제로는 '[4, 5]'가 '4::5::[]'를 줄여서 표현하는 방법이다. 어떤 방법을 사용해도 상관없지만 아무래도 '[4, 5]'가 보기가 좋다.

리스트의 첫 번째 값을 꺼낼 때는, List 모듈에 정의된 함수의 도움을 받았다. List 모듈에는 리스트에 대한 유용한 함수들이 많이 정의되어 있다. 관심있는 독자는 http://ropas.kaist.ac.kr/n/lib92/List.html에서 리스트를 다루는 함수들에는 어떤 것들이 있는지 살펴볼 수 있다.

튜플의 경우와 비슷하게 이 예제에서 'List.nth list1 3'은 실제로 정의되지 않는 값이다. 그러나 불행하게도 컴파일러가 이러한 오류를 찾아낼 수 없다는 점이 튜플의 경우와 다르다. 이러한 경우는 실행 중에 '예외상황(exception)'을 발생하게 되는데, 다행히 nML은 C++나 자바에 뒤지지 않는 훌륭한 예외상황 관리 방법을 제공하므로, 역시 큰 걱정거리가 되지는 않을 것이다.

사용자 정의 타입

nML은 풍부한 기본 타입을 제공하고 있지만, 그럼에도 불구하고 프로그램을 작성하다 보면 때로 기본 타입만으로는 뭔가 부족한 느낌이 들 때가 있다. 쉬운 예로 이진 트리를 들 수 있겠다. 이진 트리 구조는 실제로 프로그래머가 자주 사용하게 되는 구조지만, 지금까지 설명한 기본 타입만으로는 표현이 쉽지 않다. 이러한 사정은 C, C++, 자바 등 다른 프로그래밍 언어에서도 마찬가지인데, 경험있는 C/C++/자바 프로그래머라면 나름대로의 정형화된 해법을 갖고 있을 것이다.

이 단락에서는 이 문제에 대한 nML 식의 해법을 소개하려고 한다. nML의 사용자 정의 타입은 프로그래머가 일관된 방법으로 이진 트리 등의 복잡한 자료 구조까지도 깔끔하게 표현할 수 있게 해 주는 강력한 도구이다. 우선 간단한 예제로 시작해 보자.

```
# Apple ;;
  ^^^^^
오류: Apple 을(를) 모릅니다.
# type fruit = Apple | Banana | Coconut
type fruit = Apple | Banana | Coconut
# Apple ;;
val it: fruit = Apple
```

지금까지 Apple은 아무런 의미도 가지지 않았다. 그러나 앞에서처럼 fruit 타입을 정의한 이후에 Apple은 fruit 타입의 값으로 사용된다. 마찬가지로 Banana와 Coconut 역시 fruit 타입의 값이다. 여기서 한 가지 알아둘 점은 새롭게 값으로 정의하는 이름은 대문자로 시작해야 한다는 점이다. 반대로 val을 사용하여 선언하는 이름은 소문자로 시작해야 한다. 이런 것을 C/C++ 프로그램에서는 다음과 같이 표현할 수 있다.

```
typedef enum { Apple, Banana, Coconut } fruit;
```

하지만 이러한 표현은 nML에서 fruit 타입을 정의한 경우와는 많이 다르다. C의 경우는 이러한 선언의 결과로 Apple, Banana, Coconut이라는 정수 타입의 상수가 정의되고, fruit 타입의 변수는 실제로 정수형 변수와 별 차이가 없다. 따라서 C 프로그램에서 fruit 타입의 변수를 사용할 때, fruit 타입의 변수에는 항상 Apple, Banana, Coconut으로 정의된 값이 저장되어 있다는 것을 보장할 수 없는데, 때로는 이것이 프로그래머의 실수 즉 프로그램의 오류와 연관되기도 한다. C++의 타입 시스템은 C 보다 나아서 기본적으로 enum 타입을 정수 타입과 구별하지만, C++의 경우도 프로그래머가 적절히 타입을 우기면(type casting) 컴파일러는 대책이 없다.

nML의 경우 Apple은 기존의 어느 타입에도 속하지 않는 새로운 값이다. 예를 들어 정수와도 아무런 관련이 없으니 만일 프로그래머가 fruit 타입의 값을 정수로 바꾸고자 한다면, 그런 일을 하는 함수를 따로 만들어야 할 것이다. 그러나 fruit 타입의 값을 굳이 정수와 섞어쓰려 하는 것은 대개 고생을 자초하는 일이 될 뿐이다.

이와 같이 엄격한 타입 시스템은 프로그래머에게 제약으로 느껴질 수도 있으나 실제로는 얻게 되는 것이 더 많다. 조그만 제약을 감수하여 컴파일러가 타입 오류를 100% 검증해 줄 수 있게 되었는데, 이처럼 안전한 타입 시스템은 프로그램의 오류를 예방하는 데 매우 큰 도움을 주기 때문이다.

nML의 안전한 타입 시스템은 기본 타입들만을 이용해 프로그래밍을 하는 경우에도 도움을 주지만, 사용자가 새로운 타입을 정의해 사용하는 경우에 그 진가를 발휘한다. 이진 트리를 표현하는 타입을 정의해 그 진가를 확인해 보자.

지금까지와는 반대로 이번에는 C 언어의 경우를 먼저 살펴보도록 하자. 만일 C에서 이진 트리를 표현하려면 어떻게 해야 할까? 대부분의 C 프로그래머는 다음과 비슷한 타입을 선언해 이용할 것이다.

```
typedef struct _tree {
  int value;
  struct _tree *left;
  struct _tree *right;
} tree;
```

이 타입은 우리가 생각하는 이진 트리를 정확하게 표현하고 있을까? 그렇지 않다. 이와 같은 타입을 이용하는 C 프로그램에서 tree 타입의 변수가 실제로 어떤 것을 저장하게 될지는 모른다. 사실 포인터를 사용했다는 것이 가장 큰 약점인데, 그렇다고 해도 더 좋은 방법이 없으니 이렇게 하는 것이다. 이 타입 정의가 노드 간에 싸이클이 존재하는 구조를 포함하고 있다는 점도 지적할 수 있는데, 사실 프로그래머에게 더 골치 아픈 문제는 포인터 사용의 실수로 인해 발생하는 오류(dangling pointer로 인한 오류 등)일 것이다. 그러면 nML 프로그래머는 어떻게 하는가? 몇 가지 방법이 있지만, 여기서는 앞에서 예로 든 C 프로그램에서의 선언과 비슷하게 만들어 보았다.

```
# type tree = Empty
           | Tree of int * tree * tree ;;
type tree = Empty | Tree of int * tree * tree
# Empty ;;
val it: tree = Empty ;;
# Tree(0, Tree(1, Empty, Empty), Empty) ;;
val it: tree = Tree(0, Tree(1, Empty, Empty), Empty)
```

이 타입은 우리가 생각하는 이진 트리를 정확하게 표현하고 있을까? 그렇다. 이 타입에 속하면서 이진 트리가 아닌 값은 만들어낼 수 없기 때문이다. 또한 tree 타입의 값을 생성하는 방법이 얼마나 간단해 졌는지 생각해 보라. 지난 호에서 언급했듯이 메모리는 필요한 만큼만 할당되어 사용되고, 사용이 끝난 값은 자동으로 메모리에서 제거된다.

이제 조금 더 복잡한 경우를 생각해 보자. 실수에 대한 문자열 수식을 입력받아, 사칙연산을 수행하는 계산기를 작성하려고 한다. 이러한 프로그램 수행 중에 수식은 구문 분석(parsing) 단계를 거쳐 내부적으로 트리 형태로 저장된다. 사실 수식 계산과 같은 단순한 경우는 구문 분석과 계산을 동시에 할 수 있겠지만, 내부적으로 수식의 구문 분석 정보를 유지하고자 하는 경우를 가정하면, 다음과 같은 타입이 필요하다.

```
type expr = Val of real
          | Add of expr * expr
          | Sub of expr * expr
          | Mul of expr * expr
          | Div of expr * expr
```

이런 종류의 타입은 C 프로그램에서는 깔끔하게 표현하기 어렵다. C++/자바 프로그램에서는 클래스의 계층 구조를 이용해 표현할 수 있지만, 역시 같은 일을 하는 nML 프로그램만큼 간단하지는 않다.

모든 프로그래밍 언어는 각자 가장 잘 어울리는 분야가 있다. nML은 이처럼 복잡한 데이터 구조를 정의하여 사용해야 하는 작업에 매우 잘 어울리는 언어이다. nML이 가장 잘 할 수 있는 일에 기존의 프로그래밍 언어를 고집하는 것은 바람직하지 않다. 독자 여러분들은 일의 성격에 맞추어 도구를 선택할 줄 아는 현명한 프로그래머가 되기를 바란다.

패턴을 이용한 프로그래밍

필자는 패턴을 이용한 프로그래밍이 가능하다는 점을 사용자 정의 타입과 함께 nML을 다른 프로그래밍 언어와 구분해 주는 큰

장점으로 꼽는다. 그리고 패턴을 이용한 프로그래밍은 사용자 정의 타입과 매우 잘 어울린다. 패턴을 이용한 프로그래밍이라는 것이 무엇을 말하는지 우선 간단한 경우부터 살펴보도록 하자. 다음은 앞에서 사용했던 gcd 함수를 새로운 방법으로 다시 정의한 것이다.

```
# fun gcd 0 n = n
    | gcd m n = gcd (n ;
val gcd: int -> int -> int = <fun>
# gcd 9 12 ;;
val it: int = 3;
```

이 함수 정의를 원래의 정의와 비교해 보면 의미를 짐작할 수 있을 것이다. 우선 if-then-else가 사라졌다는 점이 눈에 띄는데, 이것은 패턴을 사용한 정의에 조건부 분기의 의미가 포함되어 있기 때문이다. 여기에서 gcd 함수는 두 가지로 정의되고 있는데, 함수를 호출하면 인자와 패턴이 일치하는 첫 번째 정의가 수행되는 것으로 생각하면 된다. 패턴 부분에 단순히 이름을 사용하면 모든 종류의 값과 일치되는 것으로 처리된다.

사실 우리는 인식하지 못했을 뿐 패턴을 계속 사용해 온 셈이다. val을 이용한 선언에서도 이름이 위치한다고 생각했던 곳에 실제로는 여러가지 패턴을 사용할 수 있었다. 다음을 살펴보자.

```
# val tuple = (1, 2) ;;
val tuple: int * int = (1, 2)
# val (x, y) = tuple ;;
val x: int = 1
val y: int = 2
```

이 예제에서는 튜플을 쪼개어 사용하는데 패턴을 이용했다. 앞에서 튜플을 이용한 gcd 함수를 만들어 본 적이 있는데, 사실 그때 이미 이러한 방법을 사용했던 것이다. 패턴은 리스트를 다루는 함수를 작성할 때도 유용하게 사용된다. 다음은 정수 리스트의 합을 구하는 함수를 정의한 것이다.

```
# fun sum [] = 0
    | sum (hd::tl) = hd + (sum tl) ;;
val sum: int list -> int = <fun>
# sum [1, 2, 3, 4, 5] ;;
val it: int = 15
```

이 예제에서는 리스트를 쪼개어 사용하는데 패턴을 이용했다. 앞에서 '[1, 2, 3, 4, 5]'와 '1::2::3::4::5::[]'가 동일한 표현이라고 말했던 것을 기억한다면, 이 예제에 사용된 패턴을 이해할 수 있을 것이다. 마지막으로 사용자 정의 타입과 패턴이 어떻게 어울려 사용되는지를 확인해 보도록 하자. 다음은 앞에서 정의한 expr 타입의 값을 인자로 받아 실수 값을 계산하는 함수인 eval을 정의한 것이다.

```
# fun eval Val(n) = n
    | eval Add(e1,e2) = (eval e1) + (eval e2)
    | eval Sub(e1,e2) = (eval e1) - (eval e2)
    | eval Mul(e1,e2) = (eval e1) * (eval e2)
    | eval Div(e1,e2) = (eval e1) / (eval e2) ;;
# val e = Add(Val(1.0), Mul(Val(2.0), Val(3.0))) ;;
# eval e ;;
val it: real = 7
```

이제는 모두 익숙해졌을 것 같아 중요하지 않은 출력은 생략했다. 사용자 정의 타입의 값을 처리하는 함수는 이와 같이 패턴을 이용해 정의할 수 있다. eval 함수가 정의되는 방법의 수는 expr 타입의 값을 만들어내는 방법의 수와 같다. 매우 잘 어울리면서도 효과적인 프로그래밍 방법이라고 생각되지 않는가?

C/C++/자바에 익숙한 독자들은 동일한 프로그램을 C/C++/자바로 작성한다고 생각해 보면, nML 프로그램이 상대적으로 얼마나 간단한 것인지를 실감할 수 있을 것이다.

여기까지 이해한 독자는 연습 삼아 앞에서 언급했던 fruit 타입의 값을 정수로 바꾸는 작성해 볼 수 있을 것이다. 어렵지 않다. 물론 Apple, Banana, Coconut 각각을 어떤 정수로 맵핑할 것인지는 각자의 마음대로 결정하면 된다.

예외상황 관리

앞에서 nML은 C++/자바에 뒤지지 않는 훌륭한 예외상황 관리 방법을 제공한다고 말한 바 있다. 일단 말을 꺼냈으니, nML의 예외상황 관리 방법까지는 간단하게 소개하려 한다.

예외상황 관리와 관련된 예약어는 exception, raise, handle의 세 가지가 있다. 각각을 간단히 설명하면 exception 예약어는 예외상황을 정의하는 데 사용하고, raise 예약어는 예외상황을 발생시킨다. handle 예약어는 예외상황이 발생한 경우에 수행할 일을 지정하는 데 사용한다. 이후의 예제들에서 구체적으로 이러한 예약어를 사용하는 방법을 보게 될 것이다.

정수 나눗셈에서 발생할 수 있는 Division_by_zero 예외상황은 미리 정의되어 있는 예외상황의 예다. 이 예외상황을 처리하는 예를 다음과 같이 간단히 보일 수 있다.

```
# 1 + 1 / 0 ;;
Exception: Division_by_zero.
# (1 + 1 / 0) handle Division_by_zero => 2 ;;
val it: int = 2
# 1 + (1 / 0 handle Division_by_zero => 2) ;;
val it: int = 3
```

다소 황당한 예제가 되었지만, handle 구문의 기능을 잘 보여주고 있다. handle 구문은 C++/자바의 try-catch 구문과 비슷하지만, 앞 예제와 같이 보다 세밀한 단위에도 적용할 수 있다. 계산 중에 예외상황이 발생하면, 발생한 예외상황과 일치하는 handle 구문이 선택되고, 선택된 handle 구문에서 지정한 값이 예외상황을 발생시킨 부분을 대체한다. 그 외 일치하는 handle 구문을 찾지 못한 경우의 동작 등은 C++/자바의 경우와 거의 비슷하다.

예외상황을 보다 교묘하게 이용한 프로그램의 예로 프로그램을 하나 만들어 보았다. <리스트 3>은 컴파일해 실행시키면 출력를 얻을 수 있는 완전한 프로그램이다. 핵심적인 부분은 find_even 함수인데, find_eval 함수는 인자로 받은 tree 타입의 값 즉 이진트리를 Pre-Order로 순회하면서 첫 번째 짝수를 찾아 주는 함수이다. 만일 트리에 짝수가 존재하지 않는 경우에 NotFound 예외상황을 발생시킨다. 이 함수가 결과를 얻기까지 어떤 과정을 거치는지는 독자 여러분들이 한 번 파악해 보기 바란다.

더욱 흥미진진할 nML의 세계

처음 계획은 완전한 C/C++/자바 프로그램을 같은 일을 하는 nML 프로그램과 비교하면서, nML이 잘 어울리는 일에 nML을 사용하면, 같은 프로그램이 얼마나 짧고 간단해지는가를 극명하게 보여주려는 것이었다. 실제로는 nML이라는 언어를 처음으로 소개하는 입장이었던 만큼, 부분적인 특성을 소개하는데 치중하게 되었고, 코드의 일부분만으로 비교를 진행했지만, 독자 여러분들이 조금이나마 이러한 의도를 파악하고 공감할 수 있었기를 바란다.

지금까지 nML의 가장 기본적인 것들을 간단히 소개했는데, 사실 앞으로 소개할 것이야말로 정말 흥미롭고 다른 언어에 비해 진보된 개념이다. 함수를 주고받는 함수(higher-order function), 다형 타입 함수(polymorphic function), 모듈(module), 모듈 함수(functor) 등이 이어서 소개될 것이다. nML의 이러한 개념/기능들이 어떤 경우에 유용하게 쓰이는지, 다른 언어의 유사한 기능에 비해서는 어떤 점이 더 나은지, 연재를 이어갈 실력 있는 필자들이 생생하게 소개해 줄 거라고 기대하면서, 이만 첫 번째 소개를 마친다.

<리스트 3> find_even.n

```
exception NotFound

type tree = Empty
          | Tree of int * tree * tree

fun find_even Empty = raise NotFound
  | find_even Tree(v,t1,t2) =
      if v % 2 = 0
       then v
       else (find_even t1) handle NotFound => (find_even t2)

val tree = Tree(3, Tree(1, Empty, Empty), Tree(4, Empty, Empty))

val _ = print_int (find_even tree)
```

정리 : 강경수 elegy@sbmedia.co.kr

참고자료

1. 프로그래밍 언어 nML, 프로그램 분석 시스템 연구단, http://ropas.kaist.ac.kr/n/doc/n.pdf
2. nML Programming Guide, 류석영, 이욱세, http://ropas.kaist.ac.kr/n/slide/2002.pdf
3. The Little MLer, Matthis Felleisen, Daniel P. Friedman, The MIT Press
4. Elements of ML Programming - ML97 Edition, Jeffrey D. Ullman, Prentice Hall
5. Introduction to Programming Using SML, Michael R. Hansen, Hans Rischel, Addison-Wesley