1

함수형 프로그래밍

이 장에서는 OCaml[1]을 이용한 함수형 프로그래밍(functional programming)을 소개한다. 이 책에서 함수형 프로그래밍을 다루는 이유는 크게 세 가지이다. 먼저 앞으로 프로그래밍 언어를 디자인하고 구현하게 될텐데 구현 언어로 함수형 프로그래밍 언어인 OCaml을 사용할 것이다. 프로그래밍 언어의 실행기(interpreter)나 타입 시스템(type system)과 같이 프로그램을 데이터로 다루는 프로그램을 구현하는 데 있어서 있어서 함수형 언어들이 매우 편리하기 때문이다. 두 번째 이유는 OCaml을 비롯한 함수형 언어들은 프로그래밍 언어론 관점에서 잘 설계된 언어들이라는 점이다. 이러한 언어의 특징을 살펴보는 것만으로도 프로그래밍 언어에 대한 많은 공부가 된다. 특히 함수형 프로그래밍의 개념은 최근 들어 대부분의 프로그래밍 언어들이 차용하고 있으므로 다른 언어들을 깊게 이해하는 데에도 필수적이다. 마지막으로 앞으로 우리가 만들어 갈 언어가 OCaml과 유사하다. 다음 장부터 설계할 프로그래밍 언어의 특징에 대해 미리 감을 잡을 수 있다.

1) https://ocaml.org 에서 설치할 수 있다.

1.1 OCaml 기초

OCaml로 프로그램을 작성하는 데 있어서 필요한 가장 기본적인 개념들을 익혀보자.[2)]

실행하기

OCaml 프로그램을 실행시키는 방법에는 크게 세 가지가 있다. 텍스트 편집기를 이용하여 다음과 같이 프로그램을 작성하고 `hello.ml` 이름의 파일로 저장하자.

```
let _ = print_endline "Hello World"
```

표준 출력 함수 `print_endline`을 이용하여 문자열 `"Hello World"`를 출력하는 프로그램이다.

위 프로그램을 실행하기 위해서 먼저 REPL(Read-Eval-Print Loop, 대화형 프로그램 실행 도구)을 이용할 수 있다. 명령창에서 `ocaml`을 입력하면 다음과 같이 REPL이 실행된다(`$`과 `#`을 각각 쉘과 REPL의 프롬프트를 나타내는 문자라고 하자).

```
$ ocaml
        OCaml version 4.09.0

#
```

다음과 같이 입력하고 엔터를 누르면 문자열을 출력해준다.

2) 이 절의 일부 예제는 다음 책에서 가져왔다: Yaron Minsky, Anil Madhavapeddy, Jason Hickey. Real-World OCaml. O'Reilly

```
# print_endline "Hello World";;
Hello World
- : unit = ()
```

여기에서 ;;는 REPL에게 명령을 실행하라는 의미로 사용하는 문자로, 프로그램을 텍스트 파일에 작성할 때는 적을 필요가 없다. REPL에서 직접 프로그램을 입력하지 않고 프로그램이 저장되어 있는 파일을 통째로 읽어올 수도 있다.

```
# #use "hello.ml";;
Hello World
- : unit = ()
```

REPL대신 OCaml 실행기를 직접 이용해도 된다. 다음과 같이 명령어 ocaml뒤에 실행할 파일 이름을 주면 된다.

```
$ ocaml hello.ml
Hello World
```

리눅스 환경에서 컴파일러(ocamlc)를 이용하여 실행 파일을 만들고 실행하는 방법은 다음과 같다:

```
$ ocamlc helloworld.ml
$ ls
a.out   hello.cmi  hello.cmo   hello.ml
$ ./a.out
Hello World
```

컴파일을 하면 실행파일인 a.out이 생성되며, 이를 실행하면 문자열 "Hello World"가 출력된다.

프로그램 구성의 기본단위

C, Java, Python과 같은 명령형 언어(imperative language)와 비교하여 OCaml과 같은 함수형 프로그래밍 언어가 가지는 특징 중 하나는 프로그램을 구성하는 기본 단위가 명령문(statement)이 아닌 식(expression)이라는 점이다. 일반적으로 프로그래밍 언어에서 명령문은 프로그램의 상태를 변화시키는 구문을 의미한다. 예를 들어, C나 Java의 대입문 `x = x + 1`은 메모리에 저장된 변수 `x`의 값을 1 증가시키는 명령문이다. 반면에 식은 프로그램 상태 변경없이 계산의 결과로 어떤 값을 만들어내는 구문을 뜻한다. 예를 들어, 식 `x + y`는 `x`와 `y`의 값을 참조하여 새로운 값을 계산할 뿐 메모리 상태를 변경하지는 않는다. OCaml, Haskell, Lisp과 같은 함수형 언어에서는 메모리 상태 변경 없이 식을 중심으로 값을 계산하는 형태로 프로그램을 작성하는 것이 일반적이다.

명령문 대신 식을 중심으로 프로그램을 작성하다 보니 함수형 프로그래밍에서는 문제를 푸는 절차 대신 풀고자 하는 문제를 기술하는 데 중점을 두어서 프로그램을 작성하는 것이 자연스럽다. 이처럼 프로그래밍 언어의 선택은 문제에 접근하는 사고 방식에 큰 영향을 주는데, 뒤에서 예제와 함께 좀 더 자세히 알아볼 것이다.

기본값

OCaml이 제공하는 가장 기본적인 식에는 정수, 실수, 참/거짓, 문자, 문자열 등의 기본값(primitive value)을 만들어내는 식들이 있다. 예를 들어, REPL에서 산술식(arithmetic expression) `1 + 2 *`

3을 입력해보자.

```
# 1 + 2 * 3;;
- : int = 7
```

OCaml 실행기는 입력으로 주어진 식을 계산하여 결과값이 7이라고 알려줌과 동시에 결과값의 타입(type)도 함께 알려준다. 이 경우 `int`는 결과값이 정수 타입임을 뜻한다. 실수식은 다음과 같이 계산할 수 있다.

```
# 1.1 +. 2.2 *. 3.3;;
- : float = 8.36
```

위 식의 결과값은 8.36이고 실수(`float`) 타입임을 뜻한다. OCaml은 값들을 타입에 따라 명확하게 분류하는 언어이다. 심지어 실수에 대한 덧셈과 정수에 대한 덧셈 연산자가 다르다. 위의 예에서처럼 실수에 대한 연산을 할 때에는 연산자에 `.`을 붙여야 한다. 값의 타입을 명확히 분류하지 않으면 다음과 같이 타입 오류(type error)가 발생한다.

```
# 3 + 2.0;;
Error: This expression has type float but an
expression was expected of type int
```

연산자 `+`는 피연산자로 정수값을 기대하는 데 오른쪽 피연산자의 값이 실수이므로 계산할 수 없다는 뜻이다. 정수와 실수를 더하려면 다음과 같이 명시적으로 타입을 변환해주어야 한다.

```
# 3 + (int_of_float 2.0);;
- : int = 5
```

```
# (float_of_int 3) +. 2.0;;
- : float = 5.
```

int_of_float와 float_of_int는 각각 실수를 정수로, 그리고 정수를 실수로 변환해주는 함수이다.

부울식(boolean expression)은 결과값이 참 또는 거짓인 식이다. 참과 거짓은 각각 true, false로 나타내고 이들 값의 타입은 bool이다.

```
# true;;
- : bool = true
# false;;
- : bool = false
```

산술식에 대한 비교 연산도 부울값을 만들어낸다.

```
# 1 = 2;;  (* equal to *)
- : bool = false
# 1 <> 2;;  (* not equal to *)
- : bool = true
# 2 <= (1+1);;  (* less than or equal to *)
- : bool = true
```

논리 연산자를 이용하여 기존 부울식을 엮어서 새로운 부울식을 만들 수 있다.

```
# (2 > 1) && (3 > 2 || 5 < 2);;
- : bool = true
# not (2 > 1);;
- : bool = false
```

이 외에도 OCaml에서는 문자(character), 문자열(string), 유닛(unit)값을 기본값으로 제공한다.

```
# 'c';;
- : char = 'c'
# "Objective " ^ "Caml";;
- : string = "Objective Caml"
# ();;
- : unit = ()
```

unit은 값을 하나만 가지는 타입이며 그 값은 ()로 나타낸다. C 언어에서 void 타입과 비슷한 역할을 한다고 생각하면 된다. 위의 예에서 ^는 두 문자열을 이어붙이는 연산자이다.

조건식

프로그래밍 언어가 제공하는 가장 기본적인 구문 중 하나는 조건문이다. C나 Java와 달리 OCaml에서는 조건문도 식이며, 따라서 그 결과로 값을 계산한다. 조건식(conditional expression)은 다음과 같이 생겼다.

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$$

여기서 e_1, e_2, e_3는 임의의 식을 뜻한다. 조건식의 의미는 e_1의 값이 참이면 e_2를 계산하고 e_2의 값이 거짓이면 e_3를 계산하라는 뜻이다. 예를 들어, 아래 프로그램은 e_1, e_2, e_3가 각각 2 > 1, 0, 1인 경우이다.

```
# if 2 > 1 then 0 else 1;;
- : int = 0
```

식 `2 > 1`의 값이 `true`이므로 e_2에 해당하는 식 `0`을 계산하고 그 값이 전체 조건식의 값이 된다. 조건을 다음과 같이 바꾸면 전체 조건식의 값은 e_3의 값이 된다.

```
# if 2 < 1 then 0 else 1;;
- : int = 1
```

OCaml에서 조건식을 작성할 때에는 몇가지 규칙을 지켜야 한다. 먼저 e_1은 반드시 부울식이어야 한다. 예를 들어,

```
if 1 then 2 else 3
```

는 컴파일 단계에서 타입 오류가 발생한다.

```
# if 1 then 2 else 3;;
Error: This expression has type int but an expression
was expected of type bool
```

또한 e_2와 e_3는 같은 타입의 값을 계산하는 식들이어야 한다. 예를 들어, 아래 식도 타입 오류가 발생하고 실행되지 않는다.

```
# if true then 1 else true;;
Error: This expression has type bool but an expression
was expected of type int
```

두 번째 규칙은 사실 OCaml의 정적 타입 시스템(static type system)이 요구하는 것이다. e_2와 e_3의 타입이 달라도 프로그램을 실행시키는 데는 문제가 없지만 타입을 실행 전에 결정하려다 보니

필요한 제약 사항이다. 이처럼 정적 타입 시스템을 갖춘 언어는 프로그램을 작성하는 데 있어서 요구하는 규칙이 동적 언어들에 비해 일반적으로 더 많다. **??**장에서 타입 시스템을 공부하면 이유를 이해할 수 있게 될 것이다.

변수

OCaml에서 변수는 값에 붙인 이름이다. 이름을 지으려면 키워드 `let`을 이용한다. 예를 들어,

```
# let x = 3 + 4;;
val x : int = 7
# let y = x + x;;
val y : int = 14
```

`3 + 4`의 값을 `x`로, `x + x`의 값을 `y`로 이름지었다. 이렇게 정의한 이름들은 앞으로 어디에서나 접근 가능한 일종의 전역 변수가 된다. 제한된 범위에서만 이름이 유효한 지역 변수를 정의하려면 `let...in` 식을 이용한다.

$$\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2$$

변수 x가 e_1의 값을 가지도록 정의한 후에 식 e_2를 계산하라는 뜻이다. 이 때, 변수 x의 유효범위(scope)는 e_2이다. 즉, 식 e_2안에서만 방금 정의한 변수 x가 의미를 가진다. 그리고 `let...in`도 식이므로 값을 계산해낸다. 그 값은 식 e_2의 값이다. 예를 들어, 아래 프로그램을 보자.

```
# let a = 1 in a * 2;;
- : int = 2
```

변수 a를 1로 정의한 후에 a * 2를 계산하였고 그 값이 전체식의 값이 되었다. 이 때 변수 a는 e_2에 해당하는 a * 2에서만 유효하다. 따라서 위의 let식을 계산한 후 a에 접근하면 오류가 발생한다.

```
# a;;
Error: Unbound value a
```

정의되어 있지 않은 변수 a에 접근했다는 뜻이다.

let x = e_1 in e_2의 e_1과 e_2에는 임의의 식이 올 수 있다. 다음과 같이 let을 중첩하여 사용해도 된다.

```
#  let d =
    let a = 1 in
    let b = a + a in
    let c = b + b in
      c + c;;
val d : int = 8
# d;;
- : int = 8
# a;;
Error: Unbound value a
# b;;
Error: Unbound value b
# c;;
Error: Unbound value c
```

전역 변수 d의 값을 중첩된 let을 이용하여 정의하였다. 위에서 변수 a, b, c는 변수 d의 값을 정의하기 위해서 임시로 사용한 지역 변수이므로 d가 정의된 후에는 접근할 수 없게 된다.

아래 예에서는 let의 e_1 자리에 let을 중첩하여 사용하였다.

```
# let x = (let y = 2 in y * 2) in x;;
- : int = 4
# y;;
Error: Unbound value y
```

이 경우에도 변수 y는 x를 정의하기 위하여 쓰인 지역 변수이므로 전체 식을 계산한 후에는 접근할 수 없다.

함수

함수를 정의할 때에도 let을 사용한다. 다음은 함수 square를 정의한 것이다.

```
# let square x = x * x;;
 val square : int -> int = <fun>
```

함수의 이름 다음에 함수의 인자(argument)의 이름을 적어주고, = 다음에 함수의 몸통(body)을 정의한다. 위의 함수 square는 인자 x를 받아서 결과값으로 x * x를 반환하는 함수를 정의한 것이다. 함수를 정의하면 OCaml 실행기는 그 타입을 알려주는데, 이 경우 int -> int는 정수를 인자로 받아서 정수를 되돌리는 함수라는 뜻이다. 위에서 <fun>은 정의한 값이 함수라는 뜻이다. 함수의 경우 OCaml 실행기는 구체적인 값을 보여주지는 않는다.

함수를 정의한 후에는 다음과 같이 호출하여 사용할 수 있다.

```
# square 2;;
- : int = 4
# square (2 + 5);;
```

```
- : int = 49
# square (square 2);;
- : int = 16
```

함수 square를 아래와 같이 정의할 수도 있다.

```
# let square = fun x -> x * x;;
val square : int -> int = <fun>
```

여기서 fun x -> x * x는 인자 x를 받아서 x * x를 반환하는 함수를 나타내고, 이름 square가 그 함수값을 지칭하도록 하였다. 이렇게 함수를 정의하는 방식은 앞에서 정의한 방식과 정확히 동일하다. 즉, let square x = x * x와 같은 식을 OCaml 컴파일러가 내부적으로는 let square = fun x -> x * x와 같이 처리한다고 생각하면 된다. fun x -> x * x와 같이 정의한 함수를 이름없는 함수(anonymous function)라고 부른다.

다른 예로, 인자 x가 음수인지 여부를 확인하는 함수 neg를 정의해보자. 함수의 몸통이 조건식인 경우이다.

```
# let neg x = if x < 0 then true else false;;
val neg : int -> bool = <fun>
# neg 1;;
- : bool = false
# neg (-1);;
- : bool = true
```

OCaml에서 함수 호출식은 일반적으로 e_1 e_2와 같이 생겼고, e_1과 e_2는 임의의 식이 될 수 있다. 예를 들어, 위에서 sqaure를 호출하는 세 가지 예에서 e_1은 모두 square이고 e_2는 각각 2, 2 + 5,

square 2인 경우이다. 다음과 같이 e_1에 이름없는 함수가 직접 와도 된다.

```
# (fun x -> x * x) 3;;
- : int = 9
# (fun x -> if x > 0 then x + 1 else x * x) 1;;
- : int = 2
```

함수는 여러 인자를 가질 수 있다. 예를 들어, 두 정수 x와 y를 인자로 받아서 각각의 제곱의 합을 반환하는 함수를 작성해보자.

```
# let sum_of_squares x y = (square x) + (square y);;
val sum_of_squares : int -> int -> int = <fun>
# sum_of_squares 3 4;;
- : int = 25
```

OCaml은 위 함수의 타입이 int -> int -> int라고 알려주는데 정수 두 개를 받아서 정수를 반환하는 함수 타입을 뜻한다. 또는 위 타입을 int -> (int -> int)로 해석하여 정수 인자 하나를 받아서 정수에서 정수로 가는 함수를 반환하는 함수라고 해석해도 된다. 실제로 위에서 정의한 함수를 이와 같이 해석하여 사용할 수 있다.

```
# let f = sum_of_squares 3;;
val f : int -> int = <fun>
# f 4;;
- : int = 25
```

변수 f를 sum_of_squares 3의 값으로 정의하였는데, 그 값은 정수 y를 받아서 (square 3) + (square y)의 값을 반환하는 함수가 된다. 따라서 f를 4로 호출할 수 있고 그 값은 25가 된다.

재귀 함수를 정의하는 방법도 동일하지만 `let` 다음에 키워드 `rec`을 붙여주어야 한다. 예를 들어, 함수 `factorial`을 다음과 같이 정의할 수 있다.

```
# let rec factorial a =
    if a = 1 then 1 else a * factorial (a - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

OCaml은 함수를 일급(first-class)으로 취급하는 언어이다. 함수를 정의하고 사용하는 방식이 매우 자유롭다는 뜻이다. 프로그래밍 언어에서 어떤 값이 일급이 되려면 그 값을 변수에 저장할 수 있고, 함수의 인자로 넘길 수 있고, 함수의 결과값으로 반환할 수 있어야 한다. 예를 들어, C 언어에서는 정수, 문자 등의 값들이 일급으로 취급된다. OCaml에서는 함수도 일급으로 취급되어 다음과 같이 함수가 다른 함수를 인자로 받을 수 있다.

```
# let sum f a b =
    (if f a then a else 0) + (if f b then b else 0)
val sum : (int -> bool) -> int -> int -> int = <fun>
# let even x = x mod 2 = 0;;
val even : int -> bool = <fun>
# sum even 3 4;;
- : int = 4
# sum even 2 4;;
- : int = 6
```

위에서 `even`은 인자로 받은 정수 `x`가 짝수이면 참을 반환하는 함수이다. 즉, `even`의 몸통식 `x mod 2 = 0`은 `x`를 2로 나눈 나머지가

0이면 참을 계산하는 부울식이다. 함수 sum은 인자 f, a, b를 받는데, f가 정수에서 참/거짓으로 가는 함수이다. 이 함수를 a, b에 적용하여 참이 계산될 때에만 더해서 반환하는 함수이다. 예를 들어, sum even 3 4는 함수 sum의 첫 번째 인자로 함수 even을 넘겨주었으므로 그 값은 4가 된다.

함수가 다른 함수를 반환할 수도 있다.

```
# let plus a = fun b -> a + b;;
val plus : int -> int -> int = <fun>
# let plus1 = plus 1;;
val plus1 : int -> int = <fun>
# plus1 1;;
- : int = 2
# plus1 2;;
- : int = 3
# let plus2 = plus 2;;
val plus2 : int -> int = <fun>
# plus2 1;;
- : int = 3
```

함수 plus는 인자 a를 받아서 함수 fun b -> a + b를 반환하는 함수이다. 타입 int -> int -> int은 int -> (int -> int)와 같이 해석할 수 있고, 정수 하나를 받아서 정수에서 정수로 가는 함수를 반환하는 함수 타입임을 뜻한다. 변수 plus1을 plus 1로 정의하였으므로 plus1은 인자 b를 받아서 1 + b를 반환하는 함수를 의미한다. 비슷하게 plus2는 정수 인자를 받아서 2를 더하는 함수가 된다.

sum과 plus와 같이 다른 함수를 인자로 받거나 반환하는 함수를 고차 함수(higher-order function)라고 부른다. 프로그래밍 언어

가 고차 함수를 지원하면 더 상위에서 생각하고 프로그래밍 할 수 있게 된다. 즉, 프로그래밍 패턴을 추상화하여 이름 짓는 것이 가능해져서 간결하고 읽기 쉬운 프로그램을 작성하는 데 유리하다. 고차 함수에 대해서는 1.3절에서 자세히 알아보기로 하자.

정적 타입 시스템

OCaml은 정적 타입 시스템을 갖추고 있는 언어이다. 일반적으로 프로그래밍 언어는 정적 타입 시스템을 갖춘 언어(statically typed language)와 동적 타입 시스템을 갖춘 언어(dynamically typed langauge)로 분류할 수 있는데 OCaml은 C, C++, Java, Scala 등과 함께 전자에 속한다. 이들 언어들은 타입 체킹을 정적으로, 컴파일 단계에서 수행하기 때문에 다음과 같이 타입 오류가 있는 프로그램은 컴파일러를 통과하지 못한다.

```
# 1 + true;;
Error: This expression has type bool but an
expression was expected of type int
```

반면에 Python, Lisp과 같이 동적 타입 시스템을 갖춘 언어들은 타입 체킹을 프로그램 실행 중에 수행한다. 프로그램에 타입 오류가 있는지를 미리 검사하지 않고, 실행 중에 타입이 맞지 않는 계산이 일어나는지 모니터링하는 것이다.

이들 두 부류는 상반된 장단점을 가지고 있다. 예를 들어 정적 타입 시스템을 갖춘 언어들은 타입 오류를 프로그램 개발 중에 검출할 수 있으므로 높은 안정성을 필요로 하는 프로그램 개발에 적합하다. 반면에 동적 타입 시스템을 갖춘 언어들은 정적 언어들보

다 표현의 자유도가 높고 유연하여 프로그램 개발이 빠르다는 장점이 있다.

정적 타입 시스템을 갖춘 언어들은 다시 두 가지로 구분할 수 있다. 첫 번째는 안전한(sound) 타입 시스템을 장착한 언어들이다. 이들 언어들은 프로그램 실행 중에 발생가능한 모든 타입 오류를 컴파일 단계에서 빠짐없이 찾아준다. OCaml, Haskell, Scala 등의 함수형 언어들이 주로 그렇다. 두 번째는 정적 타입 시스템을 갖추었지만 실행 중에 여전히 타입 오류가 발생할 수 있는 안전하지 않은 언어이다. C, C++ 가 대표적이다.

OCaml은 프로그램의 타입을 정적으로 체크함과 동시에 타입을 자동으로 유추해준다. 예를 들어, C나 Java에서는 다음과 같이 변수와 함수의 타입을 항상 적어주어야 했다.

```
public static int f(int n) {
  int a = 2;
  return a * n;
}
```

OCaml에서는 위 함수를 다음과 같이 타입 정보없이 정의할 수 있고, 컴파일러가 대신 타입을 자동으로 유추해준다.

```
# let f n =
    let a = 2 in
      a * n;;
val f : int -> int = <fun>
```

OCaml은 프로그램이 아무리 복잡해도 자동으로 타입을 추론할 수 있다. OCaml 실행기가 타입을 어떻게 자동으로 추론하는지 살펴보기 위해서 앞에서 정의한 함수 `sum`을 예로 들어보자.

```
# let sum f a b =
    (if f a then a else 0) + (if f b then b else 0)
val sum : (int -> bool) -> int -> int -> int = <fun>
```

OCaml은 다음의 과정을 통해 함수의 몸통식을 분석하여 타입 정보를 자동으로 유추해낸다. 먼저 조건식 `if` e_1 `then` e_2 `else` e_3에서 e_2와 e_3의 타입은 같아야 하고, `0`의 타입은 정수이므로 인자 `a`와 `b`의 타입이 `int`임을 유추할 수 있다. 그 다음에 `f`는 함수 호출의 형태(`f a` 또는 `f b`)로 사용되고 있으므로 함수 타입을 가져야 한다는 것과 `f`의 인자로 `a` 또는 `b`가 주어졌다는 점, 그리고 `f a`, `f b`의 결과값이 조건식의 e_1으로 사용되고 있다는 점에서 `f`가 함수 타입 `int -> bool`이어야 함을 추론해낸다. 마지막으로 `sum`은 덧셈의 결과를 반환하므로 그 타입이 `int`임을 추론한다. 이와 같은 과정을 거쳐서 OCaml의 자동 타입 추론(automatic type inference) 알고리즘은 프로그램 코드로부터 그 타입을 실행 전에 자동으로 추론해낸다. 자동 타입 추론의 동작 원리는 **??**장에서 자세히 살펴볼 것이다.

타입이 자동으로 유추되더라도 타입을 직접 적어주고 싶을 수 있다. 다음과 같이 하면 된다.

```
# let sum (f : int -> bool) (a : int) (b : int) : int
  = (if f a then a else 0) + (if f b then b else 0);;
val sum : (int -> bool) -> int -> int -> int = <fun>
```

이 경우 OCaml은 사용자가 적은 타입이 올바른지를 검증해준다. 예를 들어 다음과 같이 사용자가 적어준 타입이 잘못되었을 경우, 그 오류를 자동으로 찾아준다.

```
# let sum (f : int -> int) (a : int) (b : int) : int =
    (if f a then a else 0) + (if f b then b else 0);;
Error: The expression (f a) has type int but an
expression was expected of type bool
```

사용자가 실수로 f의 타입을 int -> int로 적었는데, f의 결과 타입이 bool이어야 하므로 잘못되었다는 뜻이다. OCaml의 타입 시스템은 안전(sound)하므로 사람이 적은 타입에 오류가 있다면 이를 반드시 찾아준다.

경우에 따라 어떤 식의 타입이 하나로 결정되지 않을 수도 있다. 예를 들어, 아래에 정의한 함수 id는 임의의 타입에 대해서 사용할 수 있는 함수이다.

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# id 1;;
- : int = 1
# id "abc";;
- : string = "abc"
# id true;;
- : bool = true
```

위와 같이 함수가 임의의 타입에 대해서 동작할 경우 OCaml은 'a와 같은 타입 변수를 이용하여 타입을 표현한다. 이러한 타입을 다형 타입(polymorphic type)이라 하고, 다형 타입을 가지는 함수를 다형 함수(polymorphic function)라고 부른다. 임의의 타입에 대해서 사용할 수 있는 함수라는 뜻이다.

OCaml의 다형 타입 시스템은 완전히 자유롭지는 않고 let으로 정의된 다형 함수만 지원한다. 이러한 타입 시스템을 let-다형

타입 시스템(let-polymorphic type system)이라고 부른다. 예를 들어, 아래와 같이 함수 f를 정의하면 다형 함수로 인식되어 문제 없이 실행된다.

```
# let f = fun x -> x in
    let x = f 1 in
      let y = f true in
        3;;
- : int = 3
```

하지만 동일한 의미를 가지는 프로그램을 아래와 같이 let식 없이 작성하면 타입 오류가 발생한다.

```
# (fun f ->
    let x = f 1 in
      let y = f true in
        3) (fun x -> x);;
Error: The expression has type bool but an expression
was expected of type int
```

패턴 매칭

패턴 매칭을 이용하면 중첩된 조건을 간결하게 표현할 수 있다. 예를 들어, 아래 프로그램을 보자.

```
let is123 s = if s = "1" then true
              else if s = "2" then true
              else if s = "3" then true
              else false
```

위의 함수를 패턴 매칭을 이용하여 아래와 같이 작성할 수 있다.

```
let is123 s =
  match s with
  "1" -> true
 |"2" -> true
 |"3" -> true
 | _  -> false
```

또는 다음과 같이 더 간단하게 정의할 수 있다.

```
 let is123 s =
  match s with
  "1" | "2" | "3" -> true
  | _ -> false
```

s의 값이 "a", "b", "c"중 하나이면 true, 그 외의 경우에는 false 라는 뜻이다. 패턴 매칭은 아래에서 튜플/리스트와 같은 자료형이나 사용자 정의 타입의 값을 사용하는 경우 유용하게 사용된다.

패턴 매칭을 중첩하여 사용할 경우에는 다음과 같이 괄호로 중첩된 match...with 문을 감싸주는 것이 좋다

```
match x with
| 1 -> true
| 2 ->
  (match y with
    | "a" -> true
    | "b" -> false)
| 3 -> false
```

또는 괄호 대신 begin ... end 문으로 감싸주어도 된다.

```
match x with
| 1 -> true
| 2 ->
  begin
```

```
    match y with
    | "a" -> true
    | "b" -> false
  end
| 3 -> false
```

위와 같이 하지 않으면 마지막 줄의 패턴이 `match y with`의 경우로 인식되어 타입 오류가 발생하게 된다.

튜플과 리스트

튜플(tuple)과 리스트(list)는 함수형 프로그래밍 언어에서 가장 많이 사용되는 자료 구조이다.

튜플은 값들의 묶음이다. 예를 들어, 튜플 `(1, "one")`은 정수와 문자열의 묶음이고 그 타입은 `int * string`과 같이 표현한다. 튜플 `(2, "two", true)`는 정수, 문자열, 부울값의 묶음이고 타입은 `int * string * bool`이다.

```
# let x = (1, "one");;
val x : int * string = (1, "one")
# let y = (2, "two", true);;
val y : int * string * bool = (2, "two", true)
```

튜플의 각 원소에 접근하려면 패턴 매칭을 이용하면 된다. 예를 들어, 두 원소로 구성된 튜플의 첫 번째와 두 번째 원소를 가져오는 함수 `fst`, `snd`를 다음과 같이 정의할 수 있다.

```
# let fst p = match p with (x,_) -> x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd p = match p with (_,x) -> x;;
val snd : 'a * 'b -> 'b = <fun>
```

또는 다음과 같이 함수의 인자에 튜플 패턴을 직접 사용할 수도 있다.

```
# let fst (x,_) = x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd (_,x) = x;;
val snd : 'a * 'b -> 'b = <fun>
```

타입 `'a * 'b -> 'a`는 함수 `fst`가 임의의 타입 `'a`와 `'b`로 구성된 튜플을 입력으로 받아서 `'a` 타입의 값을 반환하는 다형 함수임을 의미한다. 이와 같이 함수의 타입을 보면 함수가 하는 일을 어느정도 유추할 수 있다.

함수 인자뿐 아니라 `let`에서도 튜플 패턴을 사용할 수 있다. 예를 들어, 아래 코드를 보자.

```
# let p = (1, true);;
val p : int * bool = (1, true)
# let (x,y) = p;;
val x : int = 1
val y : bool = true
```

`p`는 튜플 `(1, true)`를 뜻하고 이를 `x`와 `y`로 분해하였다.

리스트는 같은 타입을 가지는 원소들의 나열이다. 예를 들어, 숫자 1, 2, 3으로 구성된 리스트는 `[1; 2; 3]`와 같이 표현하고 그 타입은 `int list`이다.

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
```

OCaml에서 리스트의 각 원소는 세미콜론(;)으로 구분한다. 각 원소를 콤마(,)로 구분한 리스트 [1, 2, 3]은 튜플 (1,2,3)을 원소로 가지는 리스트인 [(1,2,3)]으로 인식되므로 주의가 필요하다.

```
# [1,2,3];;
- : (int * int * int) list = [(1, 2, 3)]
# [(1,2,3)];;
- : (int * int * int) list = [(1, 2, 3)]
```

빈 리스트는 []로 나타내고 그 타입은 'a list로 다형 타입을 가진다.

```
# [];;
- : 'a list = []
```

OCaml에서 리스트의 모든 원소는 같은 타입이어야 한다. 예를 들어, [1; true]는 OCaml에서 리스트가 아니다.

```
# [1; true];;
Error: This expression has type bool but an expression
was expected of type int
```

이 제약도 정적 타입 시스템에 기인한다. 예를 들어, Python과 같이 동적 타입 시스템을 갖춘 언어에서는 리스트에 서로 다른 타입의 값이 함께 사용될 수 있다.

또한 리스트는 순서가 있는 원소들의 나열이다. 따라서 아래 두 리스트는 서로 다른 리스트이다.

```
# [1;2;3] = [2;3;1];;
- : bool = false
```

리스트의 첫 번째 원소를 머리(head), 첫 번째 원소를 제외한 나머지 리스트를 꼬리(tail)라고 부른다. 예를 들어, 리스트 `[1; 2; 3]`의 머리는 `1`, 꼬리는 `[2; 3]`이다. 일반적으로 어떤 리스트의 타입이 t `list`일 때, 머리의 타입은 t이고 꼬리의 타입은 t `list`이다.

리스트의 원소는 임의의 타입의 값이 될 수 있다. 물론 각 원소의 타입은 같아야 한다.

```
# [1;2;3;4;5];;
- : int list = [1; 2; 3; 4; 5]
# ["OCaml"; "Java"; "C"];;
- : string list = ["OCaml"; "Java"; "C"]
# [(1,"one"); (2,"two"); (3,"three")];;
- : (int * string) list =
    [(1, "one"); (2, "two"); (3, "three")]
# [[1;2;3];[2;3;4];[4;5;6]];;
- : int list list = [[1; 2; 3]; [2; 3; 4]; [4; 5; 6]]
```

마지막 예는 정수 리스트들의 리스트(`int list list`)이다. 이 때, 각 원소에 해당하는 리스트들의 길이는 모두 달라도 된다.

```
# [[1;2;3]; [4]; []];;
- : int list list = [[1; 2; 3]; [4]; []]
```

리스트 `[1;2;3]`와 `[4]`는 길이가 달라도 모두 정수 리스트이고, 빈 리스트 `[]`는 다형 타입이므로 역시 정수 리스트가 될 수 있기 때문이다.

OCaml이 제공하는 기본 리스트 연산자에는 두 가지가 있다. 먼저 `::`(cons라고 읽는다)는 리스트의 맨 앞에 원소를 하나 추가한 리스트를 만들어준다.

```
# 1::[2;3];;
- : int list = [1; 2; 3]
# 1::2::3::[];;
- : int list = [1; 2; 3]
```

두 리스트를 이어붙일 때에는 @(append라고 읽는다)을 사용한다.

```
# [1; 2] @ [3; 4; 5];;
- : int list = [1; 2; 3; 4; 5]
```

리스트를 다루는 함수를 작성할 때 패턴 매칭이 자주 쓰인다. 예를 들어, 리스트의 머리와 꼬리를 구하는 함수 hd와 tl을 정의해보자.

```
# let hd l =
    match l with
    | [] -> raise (Failure "hd is undefined")
    | a::b -> a;;
val hd : 'a list -> 'a = <fun>
# let tl l =
    match l with
    | [] -> raise (Failure "tl is undefined")
    | a::b -> b;;
val tl : 'a list -> 'a list = <fun>
# hd [1;2;3];;
- : int = 1
# tl [1;2;3];;
- : int list = [2; 3]
```

리스트의 머리와 꼬리는 빈 리스트에 대해서는 정의되지 않는다. 리스트 l이 빈 리스트가 아닌 경우이면 머리 a와 꼬리 b로 분해할 수 있고 hd는 a를, tl은 b를 반환한다(리스트 l이 원소가 하나인

리스트인 경우 꼬리 tl은 빈 리스트이다). 위의 정의에서 두 함수의 반환 타입이 다름을 눈여겨 보자. 예외 처리를 생략하면 다음과 같이 간단하게 정의할 수도 있다.

```
# let hd (a::b) = a;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val hd : 'a list -> 'a = <fun>
# let tl (a::b) = b;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val tl : 'a list -> 'a list = <fun>
```

이 경우 OCaml은 함수 hd와 tl가 빈 리스트를 고려하지 않고 있음을 알려주고 있다. 이러한 경고 메시지는 컴파일 오류는 아니지만 실행 중에 처리되지 않은 예외를 발생시킬 수 있으므로 미리 모두 제거하는 것이 좋다.

다른 예로, 리스트의 길이를 구하는 함수를 작성해보자.

```
# let rec length l =
    match l with
      [] -> 0
    |h::t -> 1 + length t;;
val length : 'a list -> int = <fun>
# length [1;2;3];;
- : int = 3
```

인자로 주어진 리스트 l이 빈 리스트이면 길이를 0으로 정의하였다. 비어 있지 않다면 l을 머리 h와 꼬리 t로 나눌 수 있고 이 경우

l의 길이는 꼬리 t의 길이에 1을 더한 것과 같아야 한다. 위의 정의에서 h는 사용하고 있지 않으므로 다음과 같이 밑줄(_)로 생략할 수 있다.

```
let rec length l =
  match l with
   [] -> 0
  |_::t -> 1 + length t
```

사용자 정의 타입

새로운 타입은 키워드 type으로 정의한다. 먼저 type을 이용하여 이미 있는 타입에 새로운 이름을 붙일 수 있다.

```
type var = string
type vector = float list
type matrix = float list list
```

또는 새로운 값의 집합을 만들고 타입으로 정의할 수 있다. 예를 들어, '요일'의 집합을 나타내는 타입 days를 다음과 같이 정의할 수 있다.

```
# type days = Mon | Tue | Wed | Thu | Fri | Sat | Sun;;
# Mon;;
- : days = Mon
# Tue;;
- : days = Tue
```

키워드 type 다음에는 정의하고자 하는 타입의 이름(days)을 적어주고, = 다음에는 그 타입에 속하는 값들을 |로 구분하여 나열하였다. Mon, ..., Sun을 생성자(constructor)라고 하고 각각 days 타입의 서로 다른 값을 의미한다. OCaml에서 타입 이름은 소문자로

시작하고 생성자는 대문자로 시작한다. 타입을 새로 정의하였으면 그 타입의 값들에 대해서 동작하는 함수를 정의할 수 있다. 예를 들어, 요일을 입력으로 받아서 다음 요일을 반환하는 함수 nextday를 다음과 같이 정의할 수 있다.

```
# let nextday d =
  match d with
  | Mon -> Tue | Tue -> Wed | Wed -> Thu | Thu -> Fri
  | Fri -> Sat  | Sat -> Sun | Sun -> Mon ;;
val nextday : days -> days = <fun>
# nextday Mon;;
- : days = Tue
```

생성자가 다른 값을 인자로 가지도록 정의할 수도 있다. 예를 들어, 사각형 또는 원을 값으로 가지는 타입 shape을 정의해보자.

```
# type shape = Rect of int * int | Circle of int;;
type shape = Rect of int * int | Circle of int
```

사각형을 의미하는 생성자 Rect는 가로와 세로 길이를 인자로 가지도록 정의하였고, 원을 의미하는 생성자 Circle은 반지름을 인자로 가지도록 하였다. 이렇게 정의한 타입에 속하는 값들의 예는 다음과 같다:

```
# Rect (2,3);;
- : shape = Rect (2, 3)
# Circle 5;;
- : shape = Circle 5
```

이렇게 정의된 도형의 넓이를 구하는 함수는 다음과 같이 작성할 수 있다.

```
# let area s =
    match s with
      Rect (w,h) -> w * h
    | Circle r -> r * r * 3;;
val area : shape -> int = <fun>
# area (Rect (2,3));;
- : int = 6
# area (Circle 5);;
- : int = 75
```

위의 예에서 편의상 원주율의 값을 3으로 근사하였다.

타입을 귀납적으로 정의하는 것도 가능하다. 예를 들어, **??**장에서 정의했던 정수 리스트의 집합을 생각해보자.

$$\frac{}{\mathsf{nil}} \qquad \frac{l}{n \cdot l}\ n \in \mathbb{Z}$$

OCaml에서 위의 집합은 아래와 같이 정의할 수 있다.

```
# type intlist = Nil | Cons of int * intlist;;
type intlist = Nil | Cons of int * intlist
```

집합의 이름(타입)을 `intlist`라고 하였고, 그 집합의 원소를 만드는 두 가지 방법을 정의하였다. 먼저 `Nil`은 빈 리스트를 뜻하는 생성자이다. `Cons`는 주어진 리스트 맨 앞에 원소를 하나 추가하여 새로운 리스트를 만드는 생성자이다. 예를 들어, 다음과 같이 리스트를 만들 수 있다.

```
# Nil;;
- : intlist = Nil
# Cons (1, Nil);;
- : intlist = Cons (1, Nil)
```

```
# Cons (1, Cons (2, Nil));;
- : intlist = Cons (1, Cons (2, Nil))
```

예를 들어, `Cons (1, Cons (2, Nil))`는 리스트 `[1;2]`를 뜻한다. 이제 리스트를 다루는 함수를 작성할 수 있다. 리스트의 길이를 구하는 함수는 다음과 같다.

```
# let rec length l =
    match l with
    | Nil -> 0
    | Cons (_, l') -> 1 + length l';;
val length : intlist -> int = <fun>
# length (Cons (1, Cons (2, Nil)));;
- : int = 2
```

다음과 같이 정의했던 정수식을 OCaml 데이터 타입으로 정의해보자.

$$\overline{n}\ n \in \mathbb{Z} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ - } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ + } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ * } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ / } E_2}$$

위 문법구조를 다음과 같이 정의할 수 있다.

```
type exp =
    Int of int
  | Minus of exp * exp
  | Plus of exp * exp
  | Mult of exp * exp
  | Div of exp * exp
```

예를 들어, `(1+2)*(3/3)`은 다음과 같이 표현된다.

```
# Mult(Plus(Int 1, Int 2), Div(Int 3, Int 3));;
- : exp = Mult (Plus (Int 1, Int 2), Div (Int 3, Int 3))
```

정수식의 의미를 나타내는 귀납 규칙은 다음과 같이 재귀 함수로 구현할 수 있다.

```
# let rec eval exp =
    match exp with
    | Int n -> n
    | Plus (e1, e2) -> (eval e1) + (eval e2)
    | Mult (e1, e2) -> (eval e1) * (eval e2)
    | Minus (e1, e2) -> (eval e1) - (eval e2)
    | Div (e1, e2) ->
      let n1 = eval e1 in
      let n2 = eval e2 in
        if n2 <> 0 then n1 / n2
        else raise (Failure "division by 0");;
val eval : exp -> int = <fun>
# eval (Mult (Plus (Int 1, Int 2),
              Div (Int 3, Int 3)));;
- : int = 3
```

의미구조 정의를 그대로 재귀 함수로 옮긴 것이다. 0으로 나누는 경우는 의미가 정의되지 않으므로 예외(exception)를 발생시켰다.

예외 처리

OCaml은 의미가 정의되지 않는 식을 계산하여 런타임 오류(run-time error)가 발생하는 경우 예외(exception)를 발생시킨다. 예를 들어, 어떤 수를 0으로 나누는 경우 다음과 같이 `Division_by_zero` 예외가 발생한다.

```
# let div a b = a / b;;
val div : int -> int -> int = <fun>
# div 10 5;;
```

```
- : int = 2
# div 10 0;;
Exception: Division_by_zero.
```

실행 중 발생하는 예외를 처리하려면 `try ... with`를 이용한다.

```
# let div a b =
    try
      a / b
    with Division_by_zero -> 0;;
val div : int -> int -> int = <fun>
# div 10 5;;
- : int = 2
# div 10 0;;
- : int = 0
```

새로운 예외를 다음과 같이 키워드 `exception`을 이용하여 정의하고 사용할 수 있다.

```
# exception Fail;;
exception Fail
# let div a b =
    if b = 0 then raise Fail
    else a / b;;
val div : int -> int -> int = <fun>
# div 10 5;;
- : int = 2
# div 10 0;;
Exception: Fail.
# try
    div 10 0
  with Fail -> 0;;
- : int = 0
```

모듈

모듈(module)은 타입과 값의 모음이다. 관련된 기능을 한 곳에 모으고 속 내용을 감추는 역할을 한다. 예를 들어, 큐(queue) 자료구조를 다음과 같이 모듈로 정의해보자.

```
module IntQueue = struct
  type t = int list
  exception E
  let empty = []
  let enq q x = q @ [x]
  let is_empty q = q = []
  let deq q =
    match q with
    | [] -> raise E
    | h::t -> (h, t)
  let rec print q =
    match q with
    | [] -> print_string "\n"
    | h::t -> print_int h; print_string " "; print t
end
```

큐를 리스트로 구현한 경우인데, 모듈의 사용자는 구현 디테일을 몰라도 다음과 같이 사용할 수 있다.

```
let q0 = IntQueue.empty
let q1 = IntQueue.enq q0 1
let q2 = IntQueue.enq q1 2
let (_,q3) = IntQueue.deq q2
let _  = IntQueue.print q1
let _  = IntQueue.print q2
let _  = IntQueue.print q3
```

큐를 만들고, 원소들을 추가 및 삭제한 후 상태를 출력하였다. 출력 결과는 아래와 같다.

```
1
1 2
2
```

지금까지 OCaml의 기본 특징들을 설명하였다. 이 정도만 알아두어도 기본적인 프로그램을 작성하는 데 무리가 없을 것이다. 다음 두 절에서는 함수형 프로그래밍에서 많이 쓰이는 두 가지 프로그래밍 스타일인 재귀 함수와 고차 함수에 대해 공부한다.

1.2 재귀 함수

함수형 프로그래밍에서는 반복을 표현할 때 반복문 대신 재귀 함수를 주로 사용한다. 재귀 함수는 반복문의 개념을 포함하는 일반적인 개념일 뿐 아니라 재귀적으로 생각하면 문제를 다른 각도에서 보다 쉽게 해결할 수 있는 경우가 많기 때문이다.

재귀 함수 연습

재귀적으로 문제를 푸는 방법을 연습해보자. 모든 재귀 함수는 다음의 구조를 가진다.

- 풀고자 하는 문제의 크기가 충분히 작은 경우에는 직접 푼다.
- 그렇지 않은 경우,
    1. 원래 문제를 동일한 구조의 부분 문제들로 쪼갠다.

2. 쪼개진 부분 문제들의 해들을 재귀적으로 구한다.

3. 부분 문제들의 해를 모아서 원래 문제의 해를 구성한다.

리스트 길이 구하기 위의 방법을 리스트 길이를 구하는 문제에 적용해보자. 먼저 빈 리스트의 경우, 그 길이는 0으로 바로 구할 수 있다. 리스트가 비어 있지 않으면 머리(head)와 꼬리(tail)로 구분할 수 있고, 꼬리의 길이를 재귀적으로 구한다. 그 길이를 n이라고 할 때, 원래 리스트의 길이는 $n+1$이 된다. 이 과정을 OCaml로 표현하면 다음과 같다.

```
let rec length l =
  match l with
  | [] -> 0
  | hd::tl -> 1 + length tl
```

리스트 이어붙이기 두 리스트를 이어붙이는 함수 `append`를 작성해보자. OCaml에서 기본 연산자 `@`로 제공하고 있지만 재귀 함수 연습을 위해 직접 정의해보자. 예를 들어, 다음과 같이 동작해야 한다.

```
# append [1; 2; 3] [4; 5; 6; 7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
# append [2; 4; 6] [8; 10];;
- : int list = [2; 4; 6; 8; 10]
```

`append`는 인자로 두 리스트 `l1`과 `l2`를 받아서 `l1@l2`에 해당하는 리스트를 반환해야 한다. 첫 번째 인자 `l1`에 대해서 재귀적으로 생각해보자. `l1`이 빈 리스트인 경우, `l2`가 두 리스트를 이어

붙인 리스트가 된다. l1이 빈 리스트가 아니라면 머리와 꼬리로 나눌 수 있고, 먼저 꼬리와 리스트 l2를 이어붙인 리스트를 재귀적으로 구한다. 그 리스트 앞에 l1의 머리를 추가하면 원래 문제의 답이 된다. OCaml로 표현하면 다음과 같다:

```
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | hd::tl -> hd::(append tl l2)
```

리스트 뒤집기 리스트를 뒤집는 함수 reverse를 작성해보자.

```
# reverse [1; 2; 3];;
- : int list = [3; 2; 1]
# reverse ["C"; "Java"; "OCaml"];;
- : string list = ["OCaml"; "Java"; "C"]
```

빈 리스트의 경우 뒤집어도 빈 리스트이다. 리스트가 비어 있지 않으면 꼬리를 먼저 뒤집고 머리를 맨 뒤에 이어붙이면 된다. OCaml로 작성하면 다음과 같다.

```
let rec reverse l =
  match l with
  | [] -> []
  | hd::tl -> (reverse tl)@[hd]
```

리스트의 n번째 원소 찾기 리스트의 n번째 원소를 반환하는 함수 nth를 작성해보자(n은 자연수라 가정한다). 주어진 리스트의 범위를 넘어서는 접근에 대해서는 예외를 발생시켜야 한다.

```
# nth [1;2;3] 0;;
- : int = 1
```

```
# nth [1;2;3] 1;;
- : int = 2
# nth [1;2;3] 2;;
- : int = 3
# nth [1;2;3] 3;;
Exception: Failure "list is too short".
```

먼저 빈 리스트에 대해서는 어떤 n에 대해서도 n번째 원소가 정의되지 않으므로 예외를 발생시킨다. 리스트가 비어 있지 않은 경우를 생각해보자. 이 경우 n이 0이면 리스트의 머리가 n번째 원소이다. n이 0이 아니면 꼬리의 $n-1$번째 원소가 원래 리스트의 n번째 원소에 해당한다. OCaml로 표현하면 다음과 같다:

```
let rec nth l n =
  match l with
  | [] -> raise (Failure "list is too short")
  | hd::tl -> if n = 0 then hd else nth tl (n-1)
```

리스트에서 처음 등장하는 특정 원소 지우기 어떤 원소를 리스트에서 제거한 리스트를 반환하는 함수를 작성해보자.

```
# remove_first 2 [1; 2; 3];;
- : int list = [1; 3]
# remove_first 2 [1; 2; 3; 2];;
- : int list = [1; 3; 2]
# remove_first 4 [1;2;3];;
- : int list = [1; 2; 3]
# remove_first [1; 2] [[1; 2; 3]; [1; 2]; [2; 3]];;
- : int list list = [[1; 2; 3]; [2; 3]]
```

인자로 주어진 리스트가 비어 있으면 제거할 원소가 없으므로 빈 리스트를 반환하면 된다. 리스트가 비어 있지 않으면 머리와 꼬리

로 나눌 수 있다. 제거하고자 하는 원소가 머리와 일치하면 꼬리를 그대로 반환하면 된다. 일치하지 않으면 꼬리에서 첫 번째로 등장하는 원소를 제거한 리스트를 구하고 머리를 그 앞에 불이면 된다. OCaml로 표현하면 다음과 같다:

```
let rec remove_first a l =
  match l with
  | [] -> []
  | hd::tl ->
    if a = hd then tl
    else hd::(remove_first a tl)
```

삽입 정렬 삽입 정렬(insertion sort)을 통해 리스트를 정렬하는 함수를 구현해보자. 먼저 정렬된 리스트와 한 원소가 주어졌을 때 그 원소의 적절한 위치를 찾아 리스트에 삽입하는 함수 `insert`를 작성해보자. 예를 들어, 다음과 같이 동작해야 한다:

```
# insert 2 [1;3];;
- : int list = [1; 2; 3]
# insert 1 [2;3];;
- : int list = [1; 2; 3]
# insert 3 [1;2];;
- : int list = [1; 2; 3]
# insert 4 [];;
- : int list = [4]
```

다음과 같이 재귀 함수로 작성할 수 있다.

```
let rec insert a l =
  match l with
  | [] -> [a]
  | hd::tl ->
```

```
    if a <= hd then a::hd::tl
    else hd::(insert a tl)
```

빈 리스트에 원소 a를 삽입하면 [a]가 된다. 비어 있지 않은 리스트 hd::tl에 원소 a를 삽입하는 과정은, 먼저 a가 리스트의 첫 번째 원소인 hd보다 같거나 작으면 hd앞에 a를 놓는 것이고 그렇지 않으면 a를 tl에 재귀적으로 삽입하고 그 앞에 hd를 놓는 것이다. 함수 insert를 정의했으면 정렬 함수 sort는 다음과 같이 간단히 정의된다.

```
let rec sort l =
  match l with
  | [] -> []
  | hd::tl -> insert hd (sort tl)
```

빈 리스트는 정렬해도 빈 리스트이다. 비어 있지 않은 리스트 hd::tl을 정렬하려면 먼저 tl을 재귀적으로 정렬한 후에 insert를 이용하여 hd를 제 위치에 넣으면 된다.

함수형 vs. 명령형 프로그래밍

위에서 정의한 삽입 정렬 함수를 C 언어에서 반복문으로 구현한 아래 코드와 비교해보자.

```
void insert_sort(int arr[], int len) {
  int i, j;
  int tmp;
  for(i = 1; i < len; i++) {
    tmp = arr[i];
    j = i - 1;
    while(j >= 0 && arr[j] > tmp) {
```

```
      arr[j + 1] = arr[j];
      j = j - 1;
    }
    arr[j + 1] = tmp;
  }
}
```

재귀 함수로 작성한 경우가 더 간결하고 가독성이 높다. 재귀 함수를 이용하는 함수형 프로그램이 더 이해하기 쉬운 이유는 문제 푸는 절차를 기술하는 대신 문제 자체를 기술하도록 유도하기 때문이다. 예를 들어, 위에서 정의한 sort의 정의를 다시 보면, 리스트가 비어 있지 않은 경우인 hd::tl을 정렬한 리스트는 tl을 정렬하고 hd를 삽입한 리스트와 일치해야 한다는 조건, 즉 정렬된 리스트가 만족해야 하는 조건을 기술하고 있다. 구현하고자 하는 함수 sort가 다음의 동치 관계를 만족해야 한다고 기술한 것이다.

```
sort (hd::tl) = insert hd (sort tl)
```

OCaml 구현에서는 위와 같이 sort가 만족해야 하는 조건을 재귀 함수로 구현한 것 뿐이다. 다른 예로 팩토리얼을 계산하는 함수 factorial은 인자로 주어진 수가 0보다 큰 경우 다음 조건을 만족해야 한다.

```
factorial n = n * factorial (n-1)
```

인자가 0인 경우와 함께 위 명세를 재귀 함수로 다음과 같이 구현할 수 있다.

```
let rec factorial n =
```

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

반면에 명령형 프로그래밍은 문제 푸는 절차를 기술하도록 유도하는 패러다임이다. 예를 들어, C 언어로 팩토리얼 함수를 작성한다면 대부분 다음과 같이 구현할 것이다.

```
int factorial (int n) {
  int i; int r = 1;
  for (i = 0; i < n; i++)
    r = r * i;
  return r;
}
```

이 코드는 위의 팩토리얼 함수가 만족해야 하는 조건을 만족하는 여러 구현 중에서 하나를 기술한 것이다. 즉, 명령형 프로그래밍에서는 문제를 기술하는 것 외에 한 단계 더 나아가 구체적인 구현 방안까지 제시해야 한다. 반면에 함수형 프로그래밍에서는 문제를 기술하는 것에 중점을 두며, 이러한 점에서 선언적 프로그래밍(declarative programming)이라 부르기도 한다.

재귀 함수의 비용

참고로 OCaml을 비롯한 함수형 프로그래밍 언어에서는 재귀 함수가 반복문에 비해 더 비싸지 않다. 예를 들어, C 에서 다음과 같은 함수를 호출하면 스택 넘침(stack overflow)이 발생하지만

```
void f() { f(); }       /* stack overflow */
```

OCaml에서는 함수가 호출될 때 메모리를 추가적으로 소모하지 않고 무한히 돈다.

```
let rec f () = f ()   (* infinite loop *)
```

두 경우 모두 함수 f는 무한히 도는 반복을 표현한 것인데, OCaml은 그 의미 그대로 실행 시켜줄 수 있는 것이다. 이 예에서 알 수 있듯이 OCaml에서는 함수가 재귀적으로 정의되었다고 해서 반복문에 비해 특별히 더 비싸지 않다. 함수 호출이 비싼 경우는 그 함수가 기술하고 있는 계산 자체가 비싼 경우이지, 단순히 계산이 재귀적으로 기술되어 있다고 해서 비싸지는 않다.

좀 더 자세히 설명하면, for나 while등의 반복문은 항상 꼬리 재귀 함수(tail-recursive function)의 형태로 변환될 수 있고, 꼬리 재귀 함수들의 호출은 비싸지 않다. 꼬리 재귀 함수란 재귀 함수를 호출한 후 더 이상 할일이 남아 있지 않은 재귀 함수를 가리킨다. 예를 들어, 리스트의 마지막 원소를 반환하는 함수 last를 생각해 보자:

```
let rec last l =
  match l with
  | [a] -> a
  | _::tl -> last tl  (* tail-recursive call *)
```

여기서 함수 호출 last tl은 꼬리 재귀 호출에 해당한다. last tl의 결과값을 계산하고 그 값을 이용해서 추가적으로 해야 할 일이 남아 있지 않다는 뜻이다. 그 결과값을 가지고 추가적으로 할 일이 없으므로 함수가 꼬리 재귀의 형태로 호출되면 호출된 지점으로 반환할 필요가 없고, 따라서 함수 호출 시 추가적인 메모리를 저장할

필요가 없다. 이러한 꼬리 호출 최적화(tail-call optimization)를 많은 함수형 언어 컴파일러들이 지원하고 있다.

반면에 팩토리얼 함수와 같이 꼬리 재귀 함수가 아닌 경우에는 재귀 함수 호출 시 추가적인 메모리가 필요하다.

```
let rec factorial n =
  if n = 1 then 1 else n * factorial (n - 1)
```

재귀 호출 `factorial (a-1)`을 계산한 후 그 결과를 가지고 할 일(a를 곱하는 일)이 남아 있으므로 꼬리 재귀 호출이 아니다. 이 경우 재귀 호출은 메모리를 소모하며 큰 수에 대해서 `factorial`을 호출하면 스택 넘침이 발생할 수 있다.

참고로 꼬리 재귀로 정의되지 않은 함수를 항상 꼬리 재귀의 형태로 변환할 수 있다. 모든 재귀 함수를 반복문으로 표현할 수 있는 것과 같은 원리이다. 예를 들어 `factorial`은 다음과 같이 꼬리 재귀의 형태로 정의할 수 있다:

```
let rec factorial n r =
  if n = 1 then r else factorial (n - 1) (r * n)
```

예를 들어, 5 팩토리얼은 `factorial 5 1`과 같이 구한다.

1.3 고차 함수

함수형 프로그래밍에서는 재귀 함수 외에도 고차 함수(higher-order function)를 많이 사용한다. 고차 함수란 다른 함수를 인자로 받거나 반환하는 함수를 뜻한다. 고차 함수는 프로그래밍 패턴을 추상

화하여 재사용 가능하게 함으로써 프로그램을 더욱 상위에서 작성할 수 있게 해 준다. 자주 사용되는 고차 함수들인 `map`, `filter`, `fold`를 중심으로 살펴보자.

map

아래 정의한 세 함수 `inc_all`, `sqaure_all`, `cube_all`은 각각 주어진 리스트의 각 원소를 증가시키고, 제곱하고, 세제곱하는 함수이다.

```
let rec inc_all l =
  match l with
  | [] -> []
  | hd::tl -> (hd+1)::(inc_all tl)

let rec square_all l =
  match l with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)

let rec cube_all l =
  match l with
  | [] -> []
  | hd::tl -> (hd*hd*hd)::(cube_all tl)
```

이들 함수들은 모두 공통된 패턴을 따라 정의되어 있다. 오직 차이점은 리스트의 각 원소에 적용하는 연산에 있다. 이 연산을 일반적으로 `f`라고 하면 위의 세 함수가 공유하고 있는 프로그래밍 패턴을 다음과 같은 고차 함수 `map`으로 표현할 수 있다.

```
let rec map f l =
```

```
match l with
| [] -> []
| hd::tl -> (f hd)::(map f tl)
```

map은 리스트 l외에 각 원소에 적용할 연산을 뜻하는 함수 f를 인자로 받아서, 리스트 l의 각 원소 a를 f a의 값으로 치환한 리스트를 생성한다. 즉, l이 리스트 [a1;a2;...;ak]일 때 map f l은 새로운 리스트 [f a1; f a2; ...; f ak]를 만들어낸다. map의 타입을 보면 이러한 의미를 어느정도 파악할 수 있다.

```
map : ('a -> 'b) -> 'a list -> 'b list
```

적용할 함수의 타입이 'a -> 'b이고, 'a 타입의 리스트를 입력으로 받아서 'b 타입의 리스트를 출력으로 만들어냄을 뜻한다. 이와 같이 일반적으로 함수의 타입은 그 함수가 하는일의 요약본(abstraction)이라 할 수 있다.

고차 함수 map을 이용하면 위의 세 함수들을 다음과 같이 간결하게 정의할 수 있다.

```
let inc_all l = map (fun x -> x + 1) l
let square_all l = map (fun x -> x * x) l
let cub_all l = map (fun x -> x * x * x) l
```

또는 공통된 인자 l을 생략하여 다음과 같이 정의해도 된다.

```
let inc_all = map (fun x -> x + 1)
let square_all = map (fun x -> x * x)
let cub_all = map (fun x -> x * x * x)
```

이를 앞의 정의와 비교해보자. map을 이용한 정의에는 각 함수의 의미가 상위 레벨에서 더욱 명확하고 직접적으로 표현되어 있음을 알 수 있다. 예를 들어, sqaure_all의 의미는 리스트 l의 각 원소에 함수 fun x -> x * x를 적용하는 것임이 직접 표현되어 있다. 반면에 map을 이용하지 않고 재귀적으로 작성한 경우에는 이러한 의미가 코드 내에 숨겨져 있고 바로 드러나지는 않는다.

잘 짜여진 프로그램이란 다른 사람이 쉽게 이해할 수 있는 프로그램이다. 구현의 세부 디테일을 모르더라도 상위 레벨에서 이해할 수 있도록 작성되어 있기 때문이다. 이런 측면에서 좋은 프로그래밍 언어란 아이디어를 보다 상위에서 표현할 수 있도록 해 주는 언어이다. 함수형 프로그래밍에서 고차 함수는 프로그램의 추상화 레벨을 높이는 데 있어서 큰 역할을 한다.

filter

아래 두 함수도 하는 일은 다르지만 동일한 패턴을 따라 정의되어 있다.

```
let rec even l =
  match l with
  | [] -> []
  | hd::tl ->
    if hd mod 2 = 0 then hd::(even tl)
    else even tl

let rec greater_than_five l =
  match l with
  | [] -> []
```

```
  | hd::tl ->
    if hd > 5 then hd::(greater_than_five tl)
    else greater_than_five tl
```

even은 리스트 l에서 짝수만 골라내고, greater_than_five는 5보다 큰 수만 골라낸다. 즉, 모두 주어진 리스트로부터 특정한 조건을 만족하는 원소들을 골라내는 함수이고, 고르는 조건만 다를 뿐이다. 이러한 공통 패턴을 다음의 고차 함수로 추상화할 수 있다.

```
let rec filter p l =
  match l with
  | [] -> []
  | hd::tl ->
    if p hd then hd::(filter p tl)
    else filter p tl
```

filter는 함수 p와 리스트 l을 받아서 l의 원소들 중에서 p를 만족하는 원소들만 모으는 일을 한다. 이때 함수 p는 결과값으로 부울값을 반환하는 함수(predicate)이어야 한다. 즉, filter의 타입은 다음과 같다.

```
filter : ('a -> bool) -> 'a list -> 'a list
```

고차 함수 filter를 이용하여 위의 두 함수를 아래와 같이 정의할 수 있다.

```
let even l = filter (fun x -> x mod 2 = 0) l
let greater_than_five l = filter (fun x -> x > 5) l
```

fold

함수형 프로그래밍에서 가장 자주 사용되는 고차 함수 가운데 하나는 fold이다. 아래 두 함수를 비교해보자.

```
let rec sum l =
  match l with
  | [] -> 0
  | hd::tl -> hd + (sum tl)

let rec prod l =
  match l with
  | [] -> 1
  | hd::tl -> hd * (prod tl)
```

두 함수 모두 주어진 리스트의 각 원소를 순회하면서 어떤 연산을 누적 적용하는 패턴을 따르고 있다. 예를 들어, 리스트 [1; 2; 3]에 대해서 위 두 함수를 적용하는 과정을 나타내면 다음과 같다.

```
sum [1; 2; 3] = 1 + (2 + (3 + 0))
prod [1; 2; 3] = 1 * (2 * (3 * 1))
```

두 함수가 하는일의 차이는 누적해서 적용할 연산과 초기값이다. sum의 경우 연산자가 +이고 초기값은 0이며, prod의 연산자는 *이고 초기값은 1이다. 이 두 가지를 추가적인 인자로 가지는 고차 함수 fold_right을 다음과 같이 정의할 수 있다.

```
let rec fold_right f l a =
  match l with
  | [] -> a
  | hd::tl -> f hd (fold_right f tl a)
```

f는 누적시킬 연산, l은 리스트, a는 초기값이다. fold_right을 이용하여 다음과 같이 함수 sum과 prod를 정의할 수 있다.

```
let sum  lst = fold_right (fun x y -> x + y) lst 0
let prod lst = fold_right (fun x y -> x * y) lst 1
```

각 함수의 의미가 더 상위에서 직접적으로 표현되었다. sum은 초기값 0으로 시작하여 리스트의 각 원소에 덧셈을 누적시켜 적용하는 함수이고, prod는 초기값 1로 시작하여 리스트의 각 원소에 곱셈을 누적시켜 적용하는 함수이다.

위의 예와 동일한 의미를 가지지만 sum과 prod가 아래와 같이 꼬리 재귀 함수로 작성되어 있는 경우를 생각해보자.

```
let rec sum a l =
  match l with
  | [] -> a
  | hd::tl -> sum (a + hd) tl

let rec prod a l =
  match l with
  | [] -> a
  | hd::tl -> prod (a * hd) tl
```

위의 두 함수의 공통패턴을 정의한 고차 함수를 fold_left라 부른다.

```
let rec fold_left f a l =
  match l with
  | [] -> a
  | hd::tl -> fold_left f (f a hd) tl
```

fold_left를 이용하여 sum과 prod를 정의하면 다음과 같다.

```
let sum a l = fold_left (fun x y -> x + y) a l
let prod a l = fold_left (fun x y -> x * y) a l
```

fold_right와 fold_left의 차이를 좀 더 자세히 살펴보자. 먼저 다음과 같이 타입이 다르다.

```
fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
fold_left  : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

더욱 중요한 차이는 연산을 누적시키는 방향이 다르다는 점이다. fold_right는 리스트의 가장 마지막 원소부터 시작해서 오른쪽에서 왼쪽으로 누적시킨다.

```
fold_right f [x;y;z] init = f x (f y (f z init))
```

반면에 fold_left는 리스트의 가장 왼쪽 원소부터 시작해서 오른쪽으로 누적시킨다.

```
fold_left f init [x;y;z] = f (f (f init x) y) z
```

따라서 적용하는 연산 f가 결합 법칙(associative law)을 만족하지 않는 경우 두 결과가 다를 수 있다.

```
# fold_right (fun x y -> x - y) [1;2;3] 0;;
- : int 2
# fold_left (fun x y -> x - y) 0 [1;2;3];;
- : int = -6
```

마지막으로 fold_left는 꼬리 재귀 함수이다. 리스트 길이가 긴 경우에는 가능하면 fold_left를 사용하는 것이 좋다.

1.4 연습 문제

문제 1 두 수 n, m $(n \leq m)$을 받아서 n이상 m이하의 수로 구성된 리스트를 반환하는 함수 range를 작성하시오.

```
range : int -> int -> int list
```

예를 들어, range 3 7 는 [3;4;5;6;7]를 만들어낸다.

문제 2 리스트의 리스트를 받아서 모든 원소들을 순서대로 포함하는 하나의 리스트를 반환하는 함수 concat을 작성하시오.

```
concat: 'a list list -> 'a list
```

예를 들어, concat [[1;2];[3;4;5]는 [1;2;3;4;5]를 만들어낸다.

문제 3 두 리스트 a와 b를 순차적으로 결합하는 함수 zipper를 작성하시오.

```
zipper: int list -> int list -> int list
```

순차적인 결합이란 리스트 a의 i번째 원소가 리스트 b의 i번째 원소 앞에 오는 것을 의미한다. 짝이 맞지 않는 원소들은 뒤에 순서대로 붙인다. 예를 들어,

```
# zipper [1;3;5] [2;4;6];;
- : int list = [1; 2; 3; 4; 5; 6]
# zipper [1;3] [2;4;6;8];;
- : int list = [1; 2; 3; 4; 6; 8]
# zipper [1;3;5;7] [2;4];;
- : int list = [1; 2; 3; 4; 5; 7]
```

문제 4 두 원소로 구성된 튜플의 리스트를 두 리스트로 분해하는 함수 unzip을 작성하시오.

```
unzip: ('a * 'b) list -> 'a list * 'b list
```

예를 들어,

```
unzip [(1,"one");(2,"two");(3,"three")]
```

은 ([1;2;3],["one";"two";"three"])을 계산한다.

문제 5 리스트 l과 정수 n을 받아서 l의 첫 n개 원소를 제외한 나머지 리스트를 구하는 함수 drop을 작성하시오.

```
drop : 'a list -> int -> 'a list
```

예를 들어,

```
drop [1;2;3;4;5] 2 = [3; 4; 5]
drop [1;2] 3 = []
drop ["C"; "Java"; "OCaml"] 2 = ["OCaml"]
```

문제 6 고차 함수 sigma를 작성하시오.

```
sigma : (int -> int) -> int -> int -> int
```

sigma f a b는 다음을 계산한다.

$$\sum_{i=a}^{b} f(i).$$

예를 들어,

```
sigma (fun x -> x) 1 10
```

는 55를,

```
sigma (fun x -> x*x) 1 7
```

는 140을 계산한다.

문제 7 고차 함수 iter를 작성하시오.

```
iter : int * (int -> int) -> (int -> int)
```

다음의 조건을 만족해야 한다.

$$\texttt{iter}(n, f) = \underbrace{f \circ \cdots \circ f}_{n}$$

$n = 0$인 경우, `iter`(n, f)는 인자를 그대로 반환하는 함수(identity function)이다. $n > 0$인 경우, `iter`(n, f)는 f를 n번 적용하는 함수이다. 예를 들어,

```
iter(n, fun x -> 2+x) 0
```

는 $2 \times n$를 계산한다.

문제 8 `fold_right`을 이용하여 고차 함수 `all`을 작성하시오.

```
all : ('a -> bool) -> 'a list -> bool
```

`all p l`은 리스트 `l`의 모든 원소들이 함수 `p`의 값을 참으로 만드는지 여부를 나타낸다. 예를 들어,

```
all (fun x -> x > 5) [7;8;9]
```

는 *true*를 계산한다.

문제 9 `fold_left`를 이용하여 정수 리스트를 숫자로 변환하는 함수를 작성하시오.

```
lst2int : int list -> int
```

예를 들어, `lst2int [1;2;3]`는 123을 계산한다. 리스트의 원소들은 0이상 9이하의 수라고 가정한다.

문제 10 아래 함수들을 `fold_right`와 `fold_left`로 다시 정의하시오.

1. ```
   let rec length l =
       match l with
       [] -> 0
       |h::t -> 1 + length t
   ```

2. ```
   let rec reverse l =
       match l with
       | [] -> []
       | hd::tl -> (reverse tl)@[hd]
   ```

3. ```
   let rec is_all_pos l =
     match l with
     | [] -> true
     | hd::tl -> (hd > 0) && (is_all_pos tl)
   ```

4. ```
   let rec map f l =
       match l with
       | [] -> []
       | hd::tl -> (f hd)::(map f tl)
   ```

5. ```
   let rec filter p l =
       match l with
   ```
   ```

```
  | [] -> []
  | hd::tl ->
    if p hd then hd::(filter p tl)
    else filter p tl
```

문제 11 귀납적으로 정의한 자연수의 집합을 생각하자.

$$\frac{}{0} \qquad \frac{n}{n+1}$$

OCaml에서는 다음과 같이 정의할 수 있다.

```
type nat = ZERO | SUCC of nat
```

예를 들어, SUCC ZERO는 1을, SUCC (SUCC ZERO)는 2를 뜻한다. 이렇게 정의된 자연수들 간의 덧셈과 곱셈을 정의하시오.

```
natadd : nat -> nat -> nat
natmul : nat -> nat -> nat
```

예를 들어,

```
# let two = SUCC (SUCC ZERO);;
val two : nat = SUCC (SUCC ZERO)
# let three = SUCC (SUCC (SUCC ZERO));;
val three : nat = SUCC (SUCC (SUCC ZERO))
# natmul two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC (SUCC ZERO)))))
# natadd two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC ZERO))))
```

문제 12 산술식 aexp를 다음과 같이 정의하자.

```
type aexp =
 | Const of int
 | Var of string
 | Power of string * int
 | Times of aexp list
 | Sum of aexp list
```

산술식과 문자를 뜻하는 문자열을 입력으로 받아서 그 문자에 대해서 미분한 산술식을 반환하는 함수를 작성하시오.

```
diff : aexp * string -> aexp
```

예를 들어, 산술식 $x^2 + 2x + 1$는 다음과 같이 표현된다.

```
Sum [Power ("x", 2); Times [Const 2; Var "x"]; Const 1]
```

이 식을 e라고 하면 diff(e, "x")는, e를 x에 대해서 미분한 $2x+2$를 반환해야 한다.

```
Sum [Times [Const 2; Var "x"]; Const 2]
```