오캐믈로 배우는 함수 프로그래밍
Ocaml Programming

박성우@포스텍

2008

머리말

이 책은 포스텍 박성우 교수님이 학부 1학년생을 위한 전자계산입문 과정에서 사용한 강의노트입니다. 함수 언어 Ocaml을 이용하여 프로그래밍 개념을 익히도록 하고 있습니다. 프로그래밍 언어론이나 컴파일러 과목에서 Ocaml을 이용하여 과제를 하는 학생들을 위하여 빌려와서 제목을 붙였습니다.

기꺼이 강의노트 사용을 허락해주신 박성우교수님께 감사드립니다.
다음은 본래의 머리말입니다.

⊦⊧⊑⊒⊥ ⊓⊔ ↦↪ □א⅄◊

이 책은 POSTECH 전자컴퓨터공학부의 EECS-101 전자계산입문 과목의 강의노트로서 EECS-101 과목의 수업 내용을 요약하였습니다. EECS-101 과목의 목표는 다음과 같습니다.

> 컴퓨터를 이용한 계산의 기본 원리와 문제해결 과정에 필수적인 계산적 사고방식을 배운다. 프로그래밍 실습을 통하여 계산적 사고방식을 어떻게 컴퓨터 프로그램으로 표현할 수 있는지를 익힌다.

즉 이 과목의 핵심주제는 계산적 사고방식computational thinking으로서 컴퓨터를 이용하여 문제를 해결할 때 가장 핵심이 되는 원리가 무엇인지를 익히는 것을 목표로 합니다. 그래서 보통 전자계산입문 과목과는 달리 프로그래밍을 수월하게 해 주지만 계산적 사고방식에는 도움이 되지 않는 반복구조문이나 계산과정과는 직접적인 연관이 없는 입출력 등의 주제는 다루지 않습니다. 그리고 프로그래밍언어를 새로 배울 때 흔히 치중하게 되는 문법에 대한 설명을 지양하고 가능하면 계산적 사고방식에 학생들이 집중할 수 있도록 작성하였습니다.

이 책의 내용에 대한 어떤 제안사항이나 의견도 환영합니다. 특히 프로그래밍에 익숙한 학생이 아니라 프로그래밍을 처음 배우는 학생들이 주는 제안사항과 의견이 본 강의노트를 향상시키는데 중요한 자료가 될 것입니다. 왜냐하면 이 책은 프로그래밍 경험이 없는 학생을 대상으로 작성하였고 따라서 프로그래밍 경험이 없는 학생이 어렵거나 이해하기 힘들다고 느끼는 부분이 더 충실하게 설명되어야 하는 부분이기 때문입니다.

이 책을 통해서 학생들이 컴퓨터를 이용한 프로그래밍이 고도의 사고과정을 요하는 흥미로운 과정이라는 것을 경험하기를 희망합니다.

차 례

1 장

연산식

컴퓨터를 이용하면 다양한 계산computation을 할 수 있다. 자연수 두개를 더하여 새로운 자연수를 찾아내는 것은 하나의 계산이며 자연수 두개를 비교하여 어느 자연수가 더 큰지를 판단하는 것도 하나의 계산이다. 이처럼 계산은 무엇을 어떻게 처리하여 결과값을 얻는지 설명하는 것으로 이루어진다. 이번 강의에서는 계산을 표현하는 기초적인 방식을 배운다.

1 자료형

계산을 수행하기 위해서 먼저 계산의 대상체를 결정해야 한다. 자료형datatype은 이러한 계산 대상체의 종류를 나타낸다. 기본적인 자료형으로 다음의 네 가지가 있다.

- 정수integer는 수학에서와 같은 개념이다. 0, 1, -1 등이 정수의 예이다.
- 부동소수floating-point number는 수학에서 다루는 실수real number를 근사적으로 표현한다. 0.0, 1.0, 3.141592 등이 부동소수의 예이다.
- 문자열character string은 연속된 문자character들로 이루어진다. “Hello”, “I am a student”, “EECS-101” 등이 문자열의 예이다.
- 불린형 값boolean value은 참과 거짓 중의 하나를 나타낸다. 보통 true와 false로 나타낸다.

각각의 자료형은 상수constant를 표현하는 방식과 기초적인 형태의 계산을 수행하는 연산자operator를 제공한다. 그리고 서로 다른 자료형을 구별하기 위해서 타입type이라는 개념이 이용된다. 아래에서는 각 자료형에서 상수와 연산자를 어떻게 이용하는지와 타입을 어떻게 이용하는지 설명한다.

2 정수

정수는 수학에서와 같은 표기법notation을 이용한다. 예를 들어 `0`, `1`, `2`는 각각 정수 0, 1, 2를 나타내는 상수연산식constant expression이다. 정수의 타입은 `int`로 표시된다.

```
# 1;;
- : int = 1
```

위의 예는 입력한 상수연산식 1이 정수 1로 해석되며 타입 int에 속하는 값으로 계산되었음을 알려준다. ;;는 연산식의 입력이 끝났음을 표시한다.

정수를 위한 연산자로는 덧셈 +, 뺄셈 -, 곱셈 *, 나눗셈 /, 나머지 계산 mod가 있다. 이들 연산자는 피연산자operand라고 불리는 두개의 정수를 받아서 새로운 정수를 넘기는 계산을 수행한다. 사용방식은 수학에서와 같이 첫번째 피연산자를 적고 연산자를 적은 뒤 두번째 피연산자를 적는다. 이러한 피연산자와 연산자의 조합을 연산식expression이라고 부른다.

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

위의 예에서 연산식 1 + 2를 입력하면 연산자 +는 상수연산식 1과 2를 피연산자로 받아서 정수 3을 계산한 뒤 타입 int에 속하는 값을 계산했음을 알려준다.

연산자는 +와 같이 피연산자를 두개 필요하는 경우도 있지만 하나만 필요로 하는 경우도 있다. 예를 들어 -는 정수 하나를 받아서 부호를 바꾼 정수를 계산하는 연산자로 쓸 수도 있다.

```
# -1;;
- : int = -1
```

위의 예에서 연산식 -1을 입력하면 연산자 -는 상수연산식 1을 피연산자로 받아서 정수 -1을 계산한 뒤 타입 int에 속하는 값을 계산했음을 알려준다. 피연산자를 두개 필요로 하는 연산자를 이진연산자binary operator라고 부르고 하나만 필요로 하는 연산자를 일진연산자unary operator라고 부른다.

일진연산자와 이진연산자는 섞어서 사용할 수 있다.

```
# 1 - -1;;
- : int = 2
```

위의 예에서 첫번째 -는 뺄셈을 위한 연산자이며 두번째 -는 부호를 바꾸는 연산자이다.

3 연산자 우선순위

여러 개의 연산자를 이용하는 연산식에서는 연산자들 간의 우선순위operator precedence를 지정해 줄 필요가 있다. 예를 들어서 3 + 5 * 2라는 연산식을 생각해보자. +와 * 중 어느 연산자를 먼저 해석하느냐에 따라서 다음과 같은 두 가지 해석이 가능하다.

왼쪽의 경우에는 +를 계산에 먼저 이용하고 오른쪽의 경우에는 *를 계산에 먼저 이용하고 있다. 실제로는 *가 +보다 우선순위가 높으며 따라서 5 * 2가 먼저 계산된다.

```
# 3 + 5 * 2;;
- : int = 13
```

일반적으로 하나의 연산식 양쪽에 우선순위가 다른 연산자가 있을 경우 우선순위가 높은 연산자를 먼저 처리한다. 정수를 위한 연산자 사이에서는 일진연산자 -가 우선순위가 가장 높고, 이진연산자 *, /, mod가 그 다음으로 우선순위가 높고, 이진연산자 +, -는 우선순위가 가장 낮다.

```
-  >  *, /, mod  >  +, -
```

따라서 위의 예에서는 5 양쪽에 +와 *라는 우선순위가 다른 연산자가 있고 *가 +보다 우선순위가 높으므로 5는 *와 결합되게 된다.

연산자 사이의 우선순위를 바꾸어서 적용하고자 하면 괄호 ()를 이용한다. 예를 들어 3 + 5 * 2에서 3 + 5를 먼저 계산하고자 하면 (3 + 5) * 2로 적으면 된다. 반대로 괄호에 포함된 연산식은 하나의 상수연산식으로 해석하면 된다. 예를 들어 (3 + 5) * 2는 3 + 5를 하나의 상수연산식으로 해석하므로 8 * 2와 동등하다.

괄호는 여러 종류의 연산자가 섞여서 사용되는 연산식을 읽기 쉽도록 해준다. 예를 들어 (2 * 2) + (3 * 3)에서 괄호는 사실 불필요하지만 2 * 2와 3 * 3을 먼저 계산한다는 것을 알려준다. 그러나 괄호를 불필요하게 많이 쓰면 연산식을 읽기가 힘들어질 수도 있으므로 주의해야 한다.

우선순위가 같은 이진연산자가 연속으로 이용되는 경우에도 계산의 순서를 구체적으로 지정해 줄 필요가 있다. 예를 들어 4 + 2 - 2의 경우 (4 + 2) - 2로 해석할 지 4 + (2 - 2)로 해석할 지 지정해 주어야 한다. 이 예는 계산순서에 무관하게 결과값이 같은 경우지만 그렇지 않은 경우도 있다. 예를 들어서 4 - 2 - 2의 경우 (4 - 2) - 2로 해석하면 결과값은 0이 되지만 4 - (2 - 2)로 해석하면 결과값은 4가 된다.

우선순위가 같은 이진연산자가 연산식 양쪽에 있을 때 계산의 순서를 정하는 규칙을 연산자의 결합성associativity라고 한다. 정수를 위한 이진연산자는 모두 왼쪽으로 결합하며 따라서 4 + 2 - 2는 (4 + 2) - 2로 해석된다. 오른쪽으로 결합하는 이진연산자의 예는 다음 절에 소개된다.

4 부동소수

부동소수는 소수점이 있는 자료형이다. 예를 들어 0.0, 1.0, 3.141592는 각각 부동소수 0.0, 1.0, 3.141592를 나타내는 상수연산식이다. 소수점 이하에 0만 있다면 생략할 수 있다. 예를 들면 1.0은 1.으로 적어도 된다. 부동소수의 타입은 float로 표시된다.

부동소수는 정수와 엄격히 구분된다. 예를 들어 0이란 숫자는 수학에서는 정수로도 해석할 수 있고 실수로도 해석할 수 있지만 연산식에서 부동소수로 나타내고자 하면 소수점을 이용하여 부동소수임을 반드시 명시하여야 한다.

```
# 0;;
- : int = 0
# 0.0;;
- : float = 0.
```

위의 예에서 `0`은 `int` 타입의 정수로 해석되지만 `0.0`은 `float` 타입의 부동소수로 해석된다.

부동소수를 위한 연산자로는 정수와 마찬가지로 부호를 바꾸는 일진연산자 `-.`와 이진연산자 덧셈 `+.`, 뺄셈 `-.`, 곱셈 `*.`, 나눗셈 `/.`이 있다. 이들 연산자의 끝에는 `.`이 반드시 붙어야 하며 이는 정수를 위한 연산자와 구별하기 위함이다. 추가로 지수승을 계산하는 이진연산자 `**`이 있다. 이들 연산자는 하나 또는 두개의 부동소수점을 받아서 새로운 부동소수점을 결과값으로 넘긴다.

정수를 위한 연산자에서처럼 일진연산자 `-.`이 우선순위가 가장 높고, `**`이 그 다음으로 우선순위가 높고, `*.`, `/.`이 그 다음이며, `+.`, `-.`이 우선순위가 가장 낮다.

```
-.  >  **  >  *.,/.  >  +.,-.
```

모든 이진연산자는 왼쪽으로 결합하지만 지수승 연산자 `**`는 오른쪽으로 결합한다.

```
# 2.0 ** 3.0 ** 2.0 ;;
- : float = 512.
# 2.0 ** (3.0 ** 2.0);;
- : float = 512.
# (2.0 ** 3.0) ** 2.0 ;;
- : float = 64.
```

위의 예에서 `3.0`의 양쪽에 같은 연산자 `**`가 있고 `**`는 오른쪽으로 결합하므로 `2.0 ** 3.0 ** 2.0`는 `2.0 ** (3.0 ** 2.0)`, 즉 `2.0 ** 9.0`으로 해석되어 512.0의 결과값을 계산한다.

부동소수를 위한 연산자는 정수를 피연산자로 받을 수 없다. 거꾸로 정수를 위한 연산자는 부동소수를 피연산자로 받을 수 없다. 정수와 부동소수를 함께 이용하고자 하면 같은 자료형으로 우선 변환해야 한다. (변환 방법은 추후에 공부한다.)

```
# 1.0 +. 1;;
This expression has type int but is here used with type float
```

위의 예에서는 `+.`는 피연산자로 두개의 부동소수를 필요로 하지만 두번째로 주어진 피연산자가 정수이기 때문에 오류메시지가 출력된다.

5 문자열

문자열은 연속된 문자들로 이루어진 자료형이다. 상수연산식은 한 쌍의 따옴표 `" "`사이에 문자들을 나열함으로써 만들어지며 문자열은 공백문자white space를 포함할 수 있다. 문자열의 타입은 `string`으로 표시된다.

```
# "Hello World!";;
- : string = "Hello World!"
```

위의 예는 입력한 상수연산식이 “Hello World!”라는 문자열로 해석되며 타입 `string`에 속하는 값으로 계산되었음을 알려준다.

문자열을 위한 연산자는 두개의 문자열을 연결해주는 이진연산자 ^가 있다.

```
# "Hello" ^ " World!";;
- : string = "Hello World!"
```

^는 오른쪽으로 결합한다.

6 불린형 값

불린형 값은 참을 나타내는 상수연산식 `true`와 거짓을 나타내는 상수연산식 `false`을 이용한다. 불린형 값의 타입은 `bool`로 표시된다.

불린형 값을 위한 연산자는 흔히 논리연산자logical operator라고 불린다. 논리연산자에는 부정negation을 계산하는 일진연산자 `not`, 결합conjunction을 계산하는 이진연산자 `&&`, 이접disjunction을 계산하는 이진연산자 `||`가 있다. 이들 연산자는 하나 또는 두개의 불린형 값을 피연산자로 받아서 새로운 불린형 값을 넘기는 계산을 수행한다. `not`은 `true`를 `false`로, `false`를 `true`로 변환한다. `&&`는 두 피연산자가 모두 `true`일 때만 `true`를 넘기며 그 외의 경우에는 `false`를 넘긴다. `||`는 두 피연산중 적어도 하나가 `true`이면 `true`를 넘기며 그 외의 경우에는 `false`를 넘긴다. `*`가 `+`보다 우선순위가 높은 것처럼 `&&`도 `||`보다 우선순위가 높다.

불린형 값은 정수나 부동소수의 대소비교를 통해서 종종 계산된다. 이를 위해서 비교연산자comparison operator라 불리는 일련의 연산자가 제공된다.

- `=`는 두 피연산자의 크기가 같으면 `true`, 그렇지 않으면 `false`를 넘긴다.
- `<>`는 `=`의 반대로서 두 피연산자의 크기가 다른 경우에 `true`를 넘긴다.
- `<`, `>`, `<=`, `>=`는 수학에서와 같은 의미로 이용된다.

이들 연산자는 정수와 부동소수, 심지어 문자열과 불린형 값에도 적용할 수 있지만 두개의 피연산자는 반드시 같은 자료형이어야 한다는 조건이 있다.

```
# 1 < 2;;
- : bool = true
# 1.0 < 2.0;;
- : bool = true
# 1.0 < 2;;
This expression has type int but is here used with type float
```

위의 예에서 수학적인 비교결과와는 상관없이 <는 첫번째 피연산자가 부동소수이므로 두번째 피연산자도 부동소수를 기대하지만 정수가 주어졌기 때문에 오류메시지를 출력하고 있다.

논리연산자와 비교연산자는 결합하면 더 복잡한 연산식을 만들 수 있다. 모든 비교연산자는 논리연산자보다 우선순위가 높기 때문에 대부분의 경우 괄호가 필요 없이 읽기 쉽게 연산식을 작성할 수 있다.

```
# 1 = 2 && 1.0 <> 2.0;;
- : bool = false
# 1 = 2 || 1.0 <> 2.0;;
- : bool = true
# not (1 = 2) && 1.0 < 2.0;;
- : bool = true
```

불린형 값은 그 자체가 최종 계산결과로 이용되기보다는 계산의 분기점에서 어느 방향을 택할지를 결정하는데 주로 이용된다. 조건연산식conditional expression은 조건식의 이러한 용도를 위해서 도입된 연산식이다.

`if` e `then` e_1 `else` e_2

위의 조건연산식에서 e는 `bool` 타입의 연산식이어야 하며 e의 계산결과가 `true`이면 e_1이 최종적으로 계산되며 그렇지 않으면 e_2가 계산된다. 여기서 주의해야할 점은 e_1과 e_2는 반드시 같은 타입을 가지는 연산식이어야 한다는 것이다. 그렇지 않으면 e의 계산결과에 따라서 서로 다른 자료형의 결과가 나올 수 있으며 이는 허용되지 않는다.

```
# if 0 < 1 then 0 else 1;;
- : int = 0
# if 0 < 1 then 0 else 1.0;;
This expression has type float but is here used with type int
```

위의 예에서 `0`과 `1.0`은 서로 다른 자료형을 가지기 때문에 오류메시지가 출력되고 있다.

조건연산식의 중요한 특징 중의 하나는 `else` 이하 부분은 반드시 필요하다는 점이다. 즉 `else` 이하는 생략된 채 `if` e `then` e_1의 형태로만 주어진 연산식은 문법에 맞지 않다. 왜냐하면 e가 `false`로 계산되면 최종 계산결과를 결정할 수가 없기 때문이다.

조건연산식에 상수연산식 `true`나 `false`가 포함되는 경우 논리연산자를 이용해서 간단하게 표현할 수가 있다. 다음은 몇 가지 예이다.

`if true then` e_1 `else` e_2	$\Longrightarrow$	e_1
`if false then` e_1 `else` e_2	$\Longrightarrow$	e_2
`if` e `then true else false`	$\Longrightarrow$	e
`if` e `then false else true`	$\Longrightarrow$	`not` e
`if` e `then` e_1 `else false`	$\Longrightarrow$	e `&&` e_1
`if` e `then true else` e_2	$\Longrightarrow$	e `||` e_2

7 계산의 오류

컴퓨터를 이용한 계산은 매우 빠르지만 상황에 따라서는 부정확한 결과가 산출되거나 계산과정이 중단되는 사태가 발생할 수도 있다. 대표적인 예로 넘침현상overflow, 정확도손실precision loss, 예외상황exception이 있다.

7.1 넘침현상

컴퓨터의 정수계산의 대부분 정확하다. 그러나 컴퓨터의 기억장치는 정보저장 용량에 한계가 있고 이는 표현할 수 있는 정수의 개수에도 한계가 있음을 의미한다. 예를 들어 정수 하나를 저장하는데 32비트bit의 기억장치를 할당하는 컴퓨터를 생각해보자. 1비트는 켜짐(on) 또는 꺼짐(off) 중의 하나의 상태를 기억할 수 있는 용량이다. 따라서 32비트는 총 2^{32}개의 서로 다른 상태를 표현할 수 있으며 이는 32비트의 용량으로는 2^{32}개의 정수만을 표현할 수 있다는 것을 의미한다. 정수를 32비트의 기억장치에 저장하는 컴퓨터는 보통 -2^{31}부터 $2^{31}-1$까지의 총 2^{32}개의 정수를 이용한다.

OCAML의 경우는 정수표현에 31비트를 사용하며 -2^{30}부터 $2^{30}-1$까지의 정수, 즉 -1073741824부터 1073741823까지의 정수만을 표현할 수 있다. 이 범위를 벗어나는 정수는 표현할 수 없으며 계산결과가 이 범위를 벗어나면 넘침현상이 발생하여 부정확한 결과가 나오게 된다. 예를 들어 표현할 수 있는 가장 큰 정수에 1을 더하면 넘침현상이 발생하여 가장 작은 정수가 계산되며, 반대로 표현할 수 있는 가장 작은 정수에 1을 빼면 넘침현상이 발생하여 가장 큰 정수가 계산된다.

```
# 1073741823 + 1;;
- : int = -1073741824
# -1073741824 - 1;;
- : int = 1073741823
```

넘침현상은 별도로 보고되지 않으므로 매우 크거나 작은 정수를 계산하는 연산식은 주의를 해서 작성하여야 한다.

7.2 정확도손실

정수의 경우와 마찬가지로 부동소수를 표현하는데 있어서도 제한된 용량의 기억장치를 이용하므로 표현할 수 있는 부동소수의 개수에는 한계가 있다. 이는 원하는 부동소수와는 약간 차이가 나는 계산결과가 생성되는 현상으로 이어지는데 이를 정확도손실이라고 한다.

```
# 0.999999999999999;;
- : float = 0.999999999999999
# 0.9999999999999999;;
- : float = 0.9999999999999999
# 0.99999999999999999;;
- : float = 1.
```

위의 예에서 첫번째 부동소수는 표현할 수 있는 부동소수 중의 하나여서 입력한데로 인식이 되지만 두번째 부동소수는 가장 가까운 부동소수로 변환되어 인식이 되고 세번째 부동소수는 1.0으로 인식되고 있다.

부동소수 표현의 한계는 부동소수 계산이 진행될 때마다 정확도손실이 발생할 수 있음을 의미한다. 예를 들어 수학적으로 x `+.` y `-.` y의 값은 x와 y의 값에 상관없이 x와 같아야 하지만 x `+.` y의 계산에서 정확도손실이 발생하면 이는 더 이상 성립하지 않는다.

```
# 2.4 +. 1.2 -. 1.2;;
- : float = 2.39999999999999947
```

위의 예에서 최종 계산결과는 2.4와는 다른데 이는 `2.4 +. 1.2`에서 이미 정확도손실이 발생했기 때문이다. 부동소수 표현의 한계로 인해서 `2.4 +. 1.2`는 3.6으로 계산되지 않고 약간 다른 부동소수로 계산되는 것이다.

```
# 2.4 +. 1.2 -. 1.2;;
- : float = 2.39999999999999947
```

부동소수 계산의 정확도손실은 결과에 약간의 차이만 있다는 점에서 정수 계산의 넘침현상보다는 덜 위험하지만, 수학적으로 적법한 등식equality이 계산상으로는 성립하지 않을 수 있음을 의미하므로 연산식을 작성할 때 주의를 해야 하는 것은 마찬가지이다. 예를 들어 복잡한 부동소수 계산결과가 수학적으로 0.0과 같은지 비교하고자 할 때 부동소수 0.0과 직접 비교하지 않고 절대값이 어떤 한계점보다 작은지를 검사하는데 이는 정확도손실을 고려한 것이다.

7.3 예외상황

넘침현상과 정확도손실은 계산의 결과에 오류가 발생하는 경우이지만 계산 자체가 중단되지는 않는다. 대조적으로 예외상황은 더 이상 계산 자체를 진행할 수가 없는 상황을 가리킨다. 대표적인 예로 정수 계산에서 0으로 나누기를 한 경우이다.

```
# 1 / 0;;
Exception: Division_by_zero.
```

위의 예에서 1을 0으로 나누는 계산은 최종 결과를 결정할 수가 없기 때문에 중단된다. (참고로 부동소수 계산에서 0.0으로 나누는 경우 예외상황은 발생하지 않고 무한대infinity를 나타내는 부동소수가 결과로 계산된다.)

넘침현상과 정확도손실과 달리 예외상황은 계산과정의 모든 가능성을 고려하지 않은 연산식 내의 명백한 오류이다. 따라서 계산중에 예외상황이 발생하지 않도록 연산식을 작성하여야 한다. (계산중에 발생하는 예외상황을 처리하는 방법이 있지만 여기서는 다루지 않는다.)

8 자료형 요약

다음 표는 이번 강의에서 다룬 자료형과 연산자를 요약하고 있다.

자료형	정수	부동소수	문자열	불린형 값
타입	`int`	`float`	`string`	`bool`
일진연산자	`-`	`-.`		`not`

이진연산자	`*`, `/`, `mod`
	`+`, `-`
		`**`
		`*.`, `/.`
		`+.`, `-.`
			`^`	`&&`
				`||`

비교연산자	`=`, `<>`, `<`, `>`, `<=`, `>=`			
조건연산식	`if` e `then` e_1 `else` e_2			

2 장

변수

계산은 보통 여러 개의 작은 계산들의 연속으로 이루어진다. 이 경우 중간 단계의 계산결과를 임시로 저장해두면 편리할 때가 있다. 이번 강의에서는 계산결과를 임시로 저장해 두는 방법을 배운다.

1 변수 정의

같은 연산식 `1 + 1`을 두 번 이용하는 계산을 생각해보자.

```
(1 + 1) + (1 + 1)
```

첫번째 `1 + 1`과 두번째 `1 + 1`을 별도로 계산하여 더하면 원하는 결과값을 얻을 수 있다. 그러나 동일한 계산이 반복되기 때문에 사실 두번째 `1 + 1`의 계산은 불필요하다. 만약 첫번째 `1 + 1`의 계산결과를 임시로 저장해 둘 수 있다면 두번째 `1 + 1`의 계산은 피할 수 있고 따라서 불필요한 계산을 없앨 수 있다. (만약 `1 + 1`이 수시간을 요하는 복잡한 계산이었다면 전체계산시간을 대폭 줄일 수 있다!)

계산의 결과를 임시로 저장해 두기 위해서는 변수variable를 이용한다. 변수는 키워드keyword `let`을 이용하여 정의한다.

`let` ⟨*변수*⟩ `=` ⟨*연산식*⟩ `;;`

이 정의는 ⟨*연산식*⟩의 계산결과를 ⟨*변수*⟩에 저장한다.

```
# let x = 1 + 1;;
val x : int = 2
```

위의 예에서 `1 + 1`의 계산결과는 변수 `x`에 저장된다. `;;`는 변수 정의가 끝났음을 표시한다. `val x : int = 2`는 변수 `x`는 `int` 타입을 가지며 `2`를 저장하고 있음을 나타내는 메시지이다.

변수이름은 영어알파벳 (`A`, `B`, ⋯, `a`, `b`, ⋯), 숫자 (`0`, `1`, ⋯), 특수문자 (`_`, `'`) 등의 자유로운 조합으로 이루어진다. 단 변수의 첫자는 반드시 영어알파벳 소문자 또는 특수문자 `_`이 되어야 한다. 다음은 적법한 변수이름의 예이다.

x, _x, x1, x2, x3, x_1, x_2, x_3, x', x'', x''', _x_1, _x_2, _x_3

xyzXYZ123abc, xyz_XYZ_123_abc

다음은 잘못된 변수이름의 예이다.

XYZ, 123xyz

첫번째는 대문자로 시작하고 두번째는 숫자로 시작하기 때문에 변수이름으로 받아들여지지 않는다. 예외적인 경우로 변수의 이름을 지정하지 않고자 하면 변수이름으로 _를 쓰면 된다. 이 경우 연산식의 결과는 저장되지 않고 버려진다.

변수는 연산식의 한 종류이며 저장된 값의 자료형을 나타내는 타입을 가진다.

```
# let x = 1 + 1;;
val x : int = 2
# x;;
- : int = 2
```

위의 예에서 변수 x는 1 + 1의 계산결과인 정수 2를 저장하도록 정의되며 이후 그 자체로 정수 2를 계산하는 int 타입을 가지는 연산식으로 이용된다.

변수는 다른 연산식의 일부로 이용될 수 있다.

```
# x + x;;
- : int = 4
```

위의 예에서 변수 x는 이미 정수 2를 저장하도록 정의되어 있으므로 연산식 x + x는 2 + 2와 동등한 계산을 수행하여 최종결과는 정수 4가 된다.

같은 이름의 변수는 새로운 값으로 재정의 할 수 있다.

```
# let x = 1 + 1;;
val x : int = 2
# x;;
- : int = 2
# let x = 2 + 2;;
val x : int = 4
# x;;
- : int = 4
```

위의 예에서 x는 1 + 1로 정의가 된 뒤 다시 2 + 2로 정의되고 있다. 일단 새로운 연산식으로 재정의되면 변수 x의 이전 값은 더 이상 이용할 수 없음을 알 수 있다. 여기서 중요한 점은 두번째 x는 첫번째 x와 무관한 새로운 변수이며 단지 같은 이름만 사용할 뿐이라는 것이다. 즉 처음에 정의한 변수 x의 값을 다시 설정하는 것이 아니라 우연히 같은 이름을 가지는 변수를 새로운 연산식을 이용하여 정의하는 것이다. 따라서 두번째 x는 첫번째 x와는 다른 자료형의 연산식으로 정의를 해도 된다.

```
# let x = 1 + 1;;
val x : int = 2
# let x = 1.0 +. 1.0;;
val x : float = 2.
```

위의 예에서 두번째 x는 첫번째 x와는 다른 타입의 변수로 정의된다.

변수를 정의할 때 연산식은 변수이름보다 먼저 처리된다.

```
# let x = 1 + 1;;
val x : int = 2
# let x = x < 4;;
val x : bool = true
```

위의 예에서 변수 x는 1 + 1의 계산결과인 정수 2로 우선 정의된다. 두번째 변수 정의에서 연산식 x < 4는 bool 타입의 변수 x를 정의하기 전에 먼저 계산되어 불린형 값 true를 결과값으로 생성한다. 만약 bool 타입의 변수 x를 먼저 정의한 뒤 연산식 x < 4를 계산하면 오류가 발생했을 것이다. 이처럼 같은 이름의 변수는 let을 이용한 변수 정의를 통해서 다른 타입의 값으로 재정의할 수 있다.

2 지역변수

let을 이용하여 정의된 변수는 같은 이름의 변수를 다시 정의할 때까지 연산식으로 남게 된다. 그러나 경우에 따라서는 특정 연산식 내에서만 임시로 사용할 수 있는 변수가 유용할 때가 있다. 예를 들어서 x라는 변수를 임시로 사용할 수 있다면 현재 같은 이름의 변수가 정의되어 있는지 여부를 일일이 기억하지 않아도 된다. 이렇게 특정 연산식 내에서만 임시로 사용할 수 있는 변수를 지역변수local variable라고 부른다.

지역변수는 키워드 let과 in을 이용하여 다음과 같이 연산식 내에 정의한다.

let ⟨*변수*⟩ = ⟨*연산식*⟩ in ⟨*최종 연산식*⟩

여기서 ⟨*연산식*⟩의 계산결과는 ⟨*변수*⟩에 임시로 저장되며 ⟨*최종 연산식*⟩의 계산에 이용된다. ⟨*변수*⟩는 지역변수이기 때문에 ⟨*최종 연산식*⟩의 계산이 끝나면 더 이상 이용할 수 없게 된다.

예를 들어 현재 x라는 변수가 정의되어 있지 않다고 가정하자.

```
# let x = 1 + 1 in x + x;;
- : int = 4
# x;;
Unbound value x
```

첫번째 연산식은 let과 in을 이용하여 지역변수 x를 정의한 뒤 최종 연산식 x + x에 이용하고 있다. 그러나 두번째 연산식에서 x는 정의되어 있지 않은 변수이므로 계산이 진행될 수 없고 따라서 오류메시지가 출력된다.

반대로 x라는 변수가 이미 정의되어 있을 때 지역변수를 정의하는 경우를 살펴보자.

```
# let x = 1.0 +. 1.0;;
val x : float = 2.
# let x = 1 + 1 in x + x;;
- : int = 4
# x;;
- : float = 2.
```

위의 예에서 float 타입을 가지는 변수 x가 먼저 정의된다. 그 다음 같은 이름의 int 타입을 가지는 지역변수 x가 정의되고 연산식 x + x의 계산에 이용된다. 이 지역변수 x는 연산식 x + x의 계산이 끝나면 감춰지게 되므로 마지막 연산식에 있는 변수 x는 처음 정의한 float 타입의 변수를 가리키게 된다.

let과 in을 이용한 연산식은 비록 지역변수를 도입하기는 하지만 여전히 계산을 표현하는 하나의 연산식이다. 따라서 연산식이 요구되는 모든 경우에 let과 in을 이용한 연산식을 쓸 수가 있다. 예를 들어 let ⟨*변수*⟩ = ⟨*연산식*⟩ in ⟨*최종 연산식*⟩ 의 ⟨*최종 연산식*⟩에 또 다른 let과 in을 이용한 연산식을 쓸 수가 있다. 이 과정을 반복하면 여러 개의 지역변수를 중첩하여 정의하는 연산식을 작성하게 된다.

let ⟨*변수*$_1$⟩ = ⟨*연산식*$_1$⟩ in
let ⟨*변수*$_2$⟩ = ⟨*연산식*$_2$⟩ in
...
let ⟨*변수*$_n$⟩ = ⟨*연산식*$_n$⟩ in
⟨*최종 연산식*⟩

다음 예는 두개의 지역변수 x와 y를 최종 연산식에 이용하고 있다.

```
# let x = 1 + 1 in
  let y = 2 + 2 in
  x + y;;
- : int = 6
```

또 다른 예로 let과 in을 이용한 연산식을 지역변수 정의하는데 이용할 수 있다. 이 경우 다음과 같은 형태의 연산식이 만들어진다.

let ⟨*변수*⟩ = let ⟨*변수'*⟩ = ⟨*연산식'*⟩ in ⟨*최종 연산식'*⟩ in ⟨*최종 연산식*⟩

다음 예는 지역변수 x를 정의하기 위해서 또 다른 let과 in을 이용한 연산식을 이용하는 경우이다.

```
# let x = let y = 1 + 1 in y + y in x + x;;
- : int = 8
```

일반적으로 let과 in을 이용한 연산식은 괄호로 묶어줌으로써 같은 타입을 가지는 연산식이 허락되는 모든 곳에 쓰일 수 있다. 예를 들면 연산식 3 = 3에서 첫번째 3 대신에 다음과 같이 let과 in을 이용한 연산식을 적어도 문제가 없다.

```
# 3 = 3;;
- : bool = true
# (let x = 1 in let y = 2 in x + y) = 3;;
- : bool = true
```

3 장

함수

복잡한 계산은 보통 같은 과정의 작은 계산을 여러번 반복한다. 이때 같은 형태의 계산이 반복될 때마다 연산식을 일일이 적어야 한다면 매우 크고 읽기 어려운 연산식이 만들어진다. 따라서 같은 형태의 계산 과정을 별도의 연산식으로 한번 적어두고 이를 반복해서 이용하는 방법이 주로 이용된다.

정수의 제곱 계산을 여러번 수행하는 계산을 생각해보자. 정수가 x가 주어질 때 x^2를 계산하는 연산식을 매번 작성하는 것은 그다지 힘들지 않다. 예를 들어 `2 * 2 + 4 * 4 + 8 * 8`은 세 정수의 제곱의 합을 계산하는 연산식이다. 그러나 정수의 네제곱을 여러번 수행하는 계산에서 일일이 연산식을 적는 것은 작성하기가 번거롭다.

```
2 * 2 * 2 * 2 + 4 * 4 * 4 * 4 + 8 * 8 * 8 * 8
```

이 경우 주어진 정수의 네제곱을 계산하는 함수function을 이용하면 간단하고 읽기 쉽게 연산식을 작성할 수 있다. 예를 들어 주어진 정수의 네제곱을 계산하는 함수가 `f`라면 위의 연산식은 아래와 같이 적을 수 있다.

```
f 2 + f 4 + f 8
```

이번 강의에서는 함수를 정의하고 이용하는 방법을 배운다.

1 함수 정의

연산식에서 이용하는 함수는 수학에서 이용하는 함수와 정확하게 같은 개념이다. 예로서 주어진 정수의 제곱을 계산하는 함수를 생각해보자. 이 함수를 연산식으로 정의하고자 하면 다음과 같이 키워드 `fun`을 이용한다.

```
fun x -> x * x
```

이 함수는 `x`라는 정수가 주어졌을 때 `x * x`라는 연산식을 계산하여 결과값으로 반환한다. 즉 `->`의 왼쪽에는 함수가 받아들이는 정수를 저장할 변수가 주어지고 오른쪽에는 주어진 정수를 이용하는 연산식이 주어진다. 여기서 변수 `x`는 함수의 형식인자formal argument라고 부르며 연산식 `x * x`는 함수의 몸통body이라고 부른다.

함수는 여러 개의 값을 받아들일 수 있으며 이때는 `->`의 왼쪽에 형식인자를 필요한 만큼 나열하면 된다.

```
fun x y -> x * y
```

위의 예는 두개의 정수 `x`와 `y`를 받아서 `x * y`를 계산하여 결과값으로 반환하는 함수이다. 세 개 이상의 정수를 받아들이는 함수의 정의도 마찬가지다.

```
fun x y z -> x * y + z
fun x y z w -> x * y + z * w
fun x y z w v -> (x * y + z * w) * v
...
```

일반적으로 n개의 형식인자를 가지는 함수는 다음과 같이 정의한다.

`fun` ⟨*형식인자*$_1$⟩ ⟨*형식인자*$_2$⟩ $\cdots$ ⟨*형식인자*$_n$⟩ `->` ⟨*몸통 연산식*⟩

함수를 정의할 때 같은 이름의 형식인자는 한번만 이용될 수 있다. 예를 들어 다음 함수는 같은 이름의 형식인자 `x`를 두번 이용하므로 잘못된 예이다.

```
fun x y x -> x + y
```

위 함수는 몸통 연산식의 변수 `x`가 첫번째 형식인자를 가리키는지 세번째 형식인자를 가리키는지 불확실하다. 그러나 몸통 연산식에서 형식인자와 같은 이름의 지역변수를 정의하는 것은 문제가 없다. 왜냐하면 몸통 연산식의 입장에서는 형식인자는 먼저 정의된 변수로 해석되기 때문이다.

```
fun x y -> let x = x + y in x + x
```

위의 예에서 형식인자 `x`와 `y`는 몸통 연산식의 입장에서는 미리 정의된 변수로 해석되므로 새로운 지역변수 `x`를 정의하는데 이용할 수 있다. 따라서 위의 함수는 두개의 정수 x와 y를 받아서 $2(x+y)$를 계산하는 함수와 동등하다.

2 함수적용

함수는 형식인자와 몸통 연산식으로 계산과정을 표현하지만 그 자체로 계산을 수행하지는 않는다. 함수를 이용하여 계산을 수행하려면 함수의 형식인자에 실제 값을 대입하는 과정이 필요하다. 이 과정을 표현하는 연산식을 함수적용function application이라고 하며 보통 함수를 호출하여 계산을 수행한다고 말한다.

함수적용은 함수 다음에 형식인자에 대입될 값을 계산하는 연산식을 나열함으로써 이루어진다.

```
(fun x -> x * x) 1
(fun x y -> x * y) (1 + 1) (2 + 2)
(fun x y z -> x * y + z) 1 1 (1 + 1)
...
```

일반적으로 n개의 형식인자를 가지는 함수를 이용하려면 다음과 같은 형태의 함수적용이 이루어져야한다.

$$\langle 함수 \rangle \langle 연산식_1 \rangle \langle 연산식_2 \rangle \cdots \langle 연산식_n \rangle$$

여기서 $\langle 연산식_i \rangle$의 계산결과는 $\langle 함수 \rangle$의 i번째 형식인자에 대입된다 ($1 \leq i \leq n$). 함수의 형식인자에 실제로 대입되는 값을 실제인자actual argument 또는 간단히 인자argument라고 부른다.

다음 예에서 `1 + 1`의 계산결과인 정수 2는 `x`에 대입되는 실제인자가 되며 `2 + 2`의 계산결과인 정수 4는 `y`에 대입되는 실제인자가 된다.

```
# (fun x y -> x * y) (1 + 1) (2 + 2);;
- : int = 8
```

따라서 이 함수적용의 결과는 `2 * 4`의 계산결과인 8이 된다.

3 이름 있는 함수

수학에서 함수를 정의할 때는 보통 함수의 이름을 함께 지정한다. 예를 들면 주어진 정수의 제곱을 계산하는 함수는 다음과 같이 구체적인 이름 f를 함께 명시한다.

$$f(x) = x * x$$

이렇게 함수 f를 정의하고 나면 $f(1), f(2), f(3), \cdots$처럼 다른 인자에 f를 여러번 적용할 수 있다.

그러나 연산식에서 이용하는 함수는 이름이 주어지지 않는다. 따라서 함수를 정의하는 데는 문제가 없지만 같은 함수를 다른 연산식에 여러번 적용하는 것은 불가능하다. 예를 들어 `fun x -> x * x`로 정의된 함수는 별도의 이름이 없기 때문에 함수적용에 쓰인다면 `(fun x -> x * x) 1`에서와 같이 단 한번 밖에 이용할 수 없다.

함수에 이름을 주는 것은 변수에 저장함으로써 가능해진다. 사실 함수는 정수나 부동소수처럼 그 자체가 하나의 값이며 따라서 변수에 저장하는데 전혀 문제가 없다.

```
# let f = fun x -> x * x;;
val f : int -> int = <fun>
# f 1;;
- : int = 1
# f 2;;
- : int = 4
```

위의 예에서 함수 `fun x -> x * x`는 변수 `f`에 저장되고 이후 두개의 실제인자 `1`과 `2`에 따로 적용되고 있다. 이 경우 편의상 `f`는 "함수 `f`"라고 부르기도 한다. (`int -> int`는 함수의 타입이며 `<fun>`은 `f`의 값이 함수임을 나타낸다. 이에 관한 내용은 5장에서 설명한다.)

```
# let f = fun x -> x * x in (f 1) + (f 2);;
- : int = 5
```

위의 예에서 함수 `fun x -> x * x`는 지역변수 `f`에 임시로 저장되어 두번 함수적용에 이용된다. `f`는 다른 지역변수처럼 연산식 밖에서는 이용할 수 없다.

함수의 몸통 연산식에 형식인자가 아닌 변수가 쓰이는 경우 그 변수의 값은 함수 정의 시점에서 유효한 같은 이름의 변수의 값을 따르며 함수 정의 이후에 도입되는 같은 이름의 변수와는 무관하다.

```
# let x = 1;;
val x : int = 1
# let f = fun y -> x + y;;
val f : int -> int = <fun>
# f 0;;
- : int = 1
# let x = -1;;
val x : int = -1
# f 0;;
- : int = 1
```

위의 예에서 함수 `f`의 몸통 연산식에는 형식인자 `y`외에 함수 밖에서 정의된 변수 `x`가 이용되고 있다. 이 변수 `x`는 함수 `f` 정의 시점에서 정수 1의 값을 가진다. 따라서 함수 `f`의 몸통 연산식은 사실 `1 + y`와 같게 되며 첫번째 함수적용 `f 0`의 결과는 1이 된다. 중요한 점은 함수 `f`의 몸통 연산식이 `x`를 `-1`로 재정의하여도 변하지 않는다는 사실이다. 따라서 두번째 함수적용 `f 0`의 결과도 여전히 1이 된다. 정리하면 함수도 하나의 값이며 정의시점에서 그 값이 결정된 뒤 다시는 변하지 않는다.

정수를 지역변수에 저장할 수 있는 것과 마찬가지로 함수도 지역변수에 저장할 수 있다.

```
# let f = fun x -> x + 1 in f 0;;
- : int = 1
# f;;
Unbound value f
```

위의 예에서 함수 `fun x -> x + 1`은 지역변수 `f`에 저장된 뒤 함수적용 `f 0`에 이용된다. 그러나 전체 연산식의 계산이 끝난 뒤에는 `f`는 참조할 수 있는 정의가 없으므로 더 이상 이용할 수 없다.

다음 예는 함수의 몸통 연산식에서 새로운 함수를 정의하고 새로운 함수의 몸통 연산식에 형식인자가 아닌 변수가 쓰이는 예이다.

```
# let f = fun a ->
    let g = fun x -> a * x in
    (g 1) + (g 2);;
val f : int -> int = <fun>
```

함수 f는 정수를 인자로 받아들여 형식인자 a에 저장한다. 함수 f의 몸통 연산식에서 새롭게 정의되는 함수 g는 형식인자 x외에 변수 a를 몸통 연산식에 이용한다. 이 변수 a는 함수 f의 적용시점에서 주어진 실제인자의 값을 가진다. 중요한 점은 함수 g는 함수 f가 적용될 때마다 새롭게 정의가 된다는 것이다. 따라서 f를 1에 적용하면 g는 fun x -> 1 * x를 저장하게 되어 최종결과는 3이 되며, 그 뒤 2에 다시 적용하면 g는 새롭게 fun x -> 2 * x를 저장하게 되어 최종결과는 6이 된다.

```
# f 1;;
- : int = 3
# f 2;;
- : int = 6
```

함수는 계산에 있어서 필수적인 요소로 자주 사용되므로 정의를 간단하게 하는 방법이 제공된다. 지금까지 배운 이름 있는 함수를 정의하는 방법은 다음과 같다.

let ⟨변수⟩ = fun ⟨형식인자$_1$⟩ ⟨형식인자$_2$⟩ $\cdots$ ⟨형식인자$_n$⟩ -> ⟨몸통 연산식⟩;;

위의 변수정의는 다음과 같이 간단하게 적을 수 있다.

let ⟨변수⟩ ⟨형식인자$_1$⟩ ⟨형식인자$_2$⟩ $\cdots$ ⟨형식인자$_n$⟩ = ⟨몸통 연산식⟩;;

즉 키워드 fun을 이용하지 않고 함수이름 다음에 형식인자를 나열하는 것이다. 지역변수에 저장하는 함수도 같은 방식으로 간단하게 정의할 수 있다. 다음은 키워드 let을 이용해서 함수를 간단하게 정의한 예이다.

```
let f x y z = x * y + z;;
let g x y z w = x * y + z * w;;
let h x y z w v = (x * y + z * w) * v;;
let f x = x + 1 in f 0;;
let f a =
    let g x = a * x in
    (g 1) + (g 2);;
```

4 함수적용의 우선순위

함수적용은 다른 연산자와 함께 쓰일 때 더 높은 우선순위가 주어진다. 예로서 정수 하나를 인자로 받는 함수 f를 두번 적용하는 연산식 f 0 + f 1을 생각해보자. 만약 덧셈 연산자 +의 우선순위가 함수적용보다 높다면 이 연산식은 f (0 + f) 1로 해석되고 결국 오류로 이어질 것이다. 그러나 함수적용은 덧셈 연산자보다 우선순위가 높기 때문에 이 연산식은 사실 (f 0) + (f 1)로 원래 의도했던 대로 해석된다. 이러한 함수적용의 우선순위를 잘 이용하면 함수적용을 많이 이용하는 연산식에서 괄호의 사용을 줄일 수 있다.

5 함수 타입

연산식으로 정의하는 함수는 정수나 부동소수처럼 그 자체가 하나의 값이며 일종의 자료형이다. 그러나 정수나 부동소수처럼 고정된 타입을 가진 것이 아니라 형식인자의 개수, 개별 형식인자의 타입, 그리고 몸통 연산식의 타입에 따라서 바뀐다.

간단한 예로 정수를 받아서 불린형 값을 계산하는 함수를 생각해보자.

```
# let f x = x > 0;;
val f : int -> bool = <fun>
```

함수 f의 타입은 `int -> bool`이며 이는 `int` 타입의 인자를 받아서 `bool` 타입의 값을 결과로 넘김을 의미한다. 즉 `->` 왼쪽에는 함수의 형식인자가 가지는 타입이 표시되고 `->` 오른쪽에는 함수의 몸통 연산식이 가지는 타입이 표시된다.

함수의 타입에는 형식인자의 개수만큼 `->`가 이용된다. 예를 들어 `int -> int -> bool`은 `int` 타입의 인자 두개를 받아서 `bool` 타입의 값을 결과로 넘기는 함수의 타입이다.

```
# let f x y = x + 1 < y - 1;;
val f : int -> int -> bool = <fun>
```

일반적으로 $T_1, T_2, \cdots, T_n$ 타입의 인자를 받아서 T 타입의 값을 결과로 넘기는 함수는 다음 타입을 가진다.

$$T_1 \texttt{ -> } T_2 \texttt{ -> } \cdots \texttt{ -> } T_n \texttt{ -> } T$$

함수를 실제인자에 적용할 때 실제인자는 함수의 타입에서 지정된 타입을 가져야 한다. 예를 들어서 `int -> bool` 타입을 가지는 함수를 `int` 타입이 아닌 `float` 타입의 연산식에 적용하는 것은 오류이다.

```
# let f x = x + x;;
val f : int -> int = <fun>
# f 0.0;;
This expression has type float but is here used with type int
```

일반적으로 T_1 `->` T_2 `->` $\cdots$ `->` T_n `->` T 타입의 함수를 적용할 때는 $T_1, T_2, \cdots, T_n$ 타입의 실제인자가 순서대로 주어져야 한다. 다음 예는 `int -> float -> bool -> bool` 타입의 함수를 올바르게 적용하는 경우와 잘못 적용하는 경우를 보여준다.

```
# let f x y z = x > 0 && y > 0.0 && z;;
val f : int -> float -> bool -> bool = <fun>
# f 1 1.0 true;;
- : bool = true
# f 1.0 1 false;;
This expression has type float but is here used with type int
```

4 장

튜플

함수는 여러 개의 인자를 받을 수 있지만 결과값은 반드시 하나만 반환할 수 있다. 그러나 많은 경우 같은 함수 내에서 여러 개의 독립된 계산을 수행한 뒤 결과를 모아서 반환할 필요가 있다. 이번 강의에서는 여러 개의 값을 하나로 묶어서 이용하는 방법을 배운다.

1 튜플 정의

두개의 정수 x와 y를 받아서 합 `x + y`와 곱 `x * y`를 계산하는 함수를 작성해보자. 합과 곱 중에서 한 가지만 계산하여 반환하는 함수를 작성하기는 쉽다.

```
let f = fun x y -> x + y;;
let g = fun x y -> x * y;;
```

그러나 모든 함수는 하나의 값만을 반환할 수 있기 때문에 지금까지 배운 연산식으로는 합과 곱을 함께 반환하는 함수를 표현할 수 없다. 이를 가능하게 하려면 두개의 값을 하나로 묶어주는 연산식이 필요하며 이때 이용하는 연산식이 튜플tuple이다.

튜플은 순서쌍의 일반화된 형태인 자료형으로서 다음과 같이 연산식을 괄호 안에 나열함으로써 생성된다.

$$(\langle \text{연산식}_1 \rangle, \langle \text{연산식}_2 \rangle, \cdots, \langle \text{연산식}_n \rangle)$$

위의 연산식의 계산결과는 $\langle$연산식$_i\rangle$를 독립적으로 계산한 결과를 순서대로 모은 값이 된다. ($1 \leq i \leq n$). 여기서 튜플 내에 들어가는 연산식들은 같은 타입을 가질 필요는 없다. 만약 $\langle$연산식$_i\rangle$의 타입이 T_i이면 위 튜플은 다음의 튜플타입tuple type을 가진다.

$$T_1 * T_2 * \cdots * T_n$$

여기서 *는 정수의 곱셈연산자가 아니며 튜플타입에서 개별 타입을 연결시켜주는 역할을 수행한다.

다음은 튜플 연산식의 예이다.

```
# (1, true, 1.0);;
- : int * bool * float = (1, true, 1.)
# (1 + 1, true && false, 1.0 +. 1.0);;
- : int * bool * float = (2, false, 2.)
```

처음 튜플에는 상수연산식만 이용되며 두번째 튜플에서는 추가 계산이 필요한 연산식이 이용되고 있다. 그러나 두 경우 모두 정수, 불린형 값, 부동소수가 순서대로 계산되어 하나로 묶이므로 타입은 똑같이 `int * bool * float`가 된다.

```
# ((let x = 1 in x + x), x + x);;
Unbound value x
```

위의 예는 튜플에 나열되는 연산식들이 독립적으로 계산됨을 보여준다. 즉 첫번째 연산식 `let x = 1 in x + x`에서 이용하는 지역변수 `x`는 연산식 밖에서는 보이지 않으므로 두번째 연산식의 계산에서 오류메시지가 출력된다.

튜플은 일종의 연산식이기 때문에 다른 큰 튜플 내에서 이용하는데 문제가 없다.

```
# ((1, true), (1.0, false));;
- : (int * bool) * (float * bool) = ((1, true), (1., false))
```

위의 연산식은 두개의 튜플 `(1, true)`와 `(1.0, false)`로 이루어진 튜플이다. 첫번째 튜플은 `int * bool`의 타입을 가지고 두번째 튜플은 `float * bool`의 타입을 가지므로 전체 튜플은 `(int * bool) * (float * bool)`의 타입을 가진다. 여기서 주의할 점은 위 튜플은 네 개의 연산식으로 이루어진 `(1, true, 1.0, false)`와는 다른 튜플이라는 점이다. 따라서 `(int * bool) * (float * bool)`도 `int * bool * float * bool`과는 다른 타입이다.

일반 연산식의 계산결과를 변수에 저장할 수 있는 것처럼 튜플 연산식의 계산결과도 변수에 저장할 수 있다.

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

위의 예에서 변수 `x`는 튜플 `(1, 1)`을 저장하며 변수 `y`는 같은 변수 `x`를 두번 이용하여 생성되는 튜플을 저장한다.

두개 이상의 값을 하나로 묶는 연산식인 튜플을 이용하면 다음과 같이 처음에 작성하고자 했던 함수를 표현할 수 있다.

```
# let h = fun x y -> (x + y, x * y);;
val h : int -> int -> int * int = <fun>
```

함수 f의 타입은 int 타입의 인자를 두개 받아서 int * int 타입의 값을 넘긴다는 것을 알려주고 있다. 즉 int -> int -> int * int는 int -> int -> (int * int)로 해석된다.

튜플에서 괄호는 사실 연산식을 읽기 쉽게 하기 위해서만 사용되며 반드시 필요한 것은 아니다. 그러나 연산식을 읽기 쉽게 하기 위해서 괄호를 항상 사용하는 것을 권장한다. 또한 괄호 없이 튜플을 만들 수 있다는 것을 기억해 둘 필요가 있다.

```
# (let x = 1 in x + x, x + x);;
- : int * int = (2, 2)
```

위의 예에서 x + x, x + x는 사실 괄호 없이 만들어지는 튜플이다. 따라서 위의 연산식은 ((let x = 1 in x + x), x + x)가 아니라 (let x = 1 in (x + x, x + x))로 해석되기 때문에 정상적으로 계산되어 (2, 2)의 결과가 생성된다.

2 패턴검사

튜플은 여러 개의 값을 하나로 묶는데 반드시 필요하지만 개별적인 값들을 다시 찾아내는 방법이 없다면 결국은 쓸모가 없다. 이는 함수를 정의하는 방법만 있고 적용하는 방법이 없다면 모든 함수는 결국 쓸모가 없는 이치와 같다. 따라서 튜플을 다시 분리하여 개별적인 값들을 찾아내는 방법이 필요하다. 이는 튜플이라는 자료형의 연산자로 생각할 수 있다.

n개의 값으로 이루어진 튜플은 n개의 변수를 동시에 정의하는 형식으로 분리할 수 있다.

$$\texttt{let (}\langle \text{변수}_1 \rangle, \langle \text{변수}_2 \rangle, \cdots, \langle \text{변수}_n \rangle\texttt{) = } \langle \text{연산식} \rangle \texttt{ ;;}$$

여기서 $\langle$연산식$\rangle$의 계산 결과는 n개의 값을 가지는 튜플이어야 하며 각각의 값은 순서대로 $\langle$변수$_1\rangle$, $\langle$변수$_2\rangle$, $\cdots$, $\langle$변수$_n\rangle$에 저장된다. 따라서 ($\langle$변수$_1\rangle$, $\langle$변수$_2\rangle$, $\cdots$, $\langle$변수$_n\rangle$)는 연산식이 아닌 변수의 일반화된 형태이며 패턴pattern이라고 부른다. 그리고 패턴을 이용하여 동시에 여러 개의 변수를 정의하는 것을 패턴검사pattern matching라고 부른다. 거꾸로 하나의 변수도 패턴의 특수한 형태라고 볼 수 있다.

다음은 패턴검사를 이용하여 튜플을 구성하는 값들을 변수에 저장하는 예이다.

```
# let (x, y, z) = (1, true, 1.0);;
val x : int = 1
val y : bool = true
val z : float = 1.
```

위의 예에서 1은 변수 x에, true는 변수 y에, 1.0은 변수 z에 순서대로 저장됨을 알 수 있다. 만약 튜플의 형태와 같지 않은 패턴이 주어지면 다음과 같이 오류메시지가 발생한다.

```
# let (x, y) = (1, true, 1.0);;
This expression has type int * bool * float but is here used with type
  'a * 'b
```

여기서 =의 오른쪽에 있는 튜플은 세 개의 값을 포함하지만 왼쪽에 있는 패턴은 두개의 변수만을 포함하기 때문에 패턴검사가 성공하지 못하고 있다. (위의 에러메시지에 있는 'a * 'b의 의미는 추후에 배운다.)

튜플이 더 큰 튜플에 포함될 수 있는 것처럼 패턴도 더 큰 패턴에 포함될 수 있다. 예를 들어 ((1, true), (1.0, false))를 생각해보자. 이 튜플은 두개의 작은 튜플로 이루어지기 때문에 두개의 변수로 이루어진 패턴을 이용하여 분리할 수 있다.

```
# let (x, y) = ((1, true), (1.0, false));;
val x : int * bool = (1, true)
val y : float * bool = (1., false)
```

여기서 x와 y도 역시 2개의 값으로 이루어지는 튜플을 저장하므로 패턴검사를 반복하여 분리할 수 있다.

```
# let (x1, x2) = x;;
val x1 : int = 1
val x2 : bool = true
# let (y1, y2) = y;;
val y1 : float = 1.
val y2 : bool = false
```

그러나 패턴을 포함하는 패턴을 이용하면 한번의 패턴검사로 x1, x2, y1, y2에 값을 저장할 수 있다.

```
# let ((x1, x2), (y1, y2)) = ((1, true), (1.0, false));;
val x1 : int = 1
val x2 : bool = true
val y1 : float = 1.
val y2 : bool = false
```

패턴검사를 이용하면 튜플을 넘기는 함수의 결과값을 유용하게 쓸 수 있다.

```
# let h = fun x y -> (x + y, x * y);;
val h : int -> int -> int * int = <fun>
# let (a, b) = h 1 2;;
val a : int = 3
val b : int = 2
```

위의 예에서 함수적용 h 1 2의 결과로 넘어오는 튜플은 바로 패턴 (a, b)와 검사되어 변수 a와 b에 각각의 값을 저장한다.

일반적으로 변수가 정의되는 모든 곳에 패턴이 쓰일 수 있다. 예를 들면 함수의 형식인자로 변수가 쓰이므로 함수의 형식인자로 패턴을 사용해도 된다.

```
# let f = fun (x, y) (v, w) -> x * v + y * w;;
val f : int * int -> int * int -> int = <fun>
```

위의 함수 f는 두개의 값으로 이루어지는 튜플을 두개 받아서 대응되는 값끼리의 곱을 합한다. 만약 형식인자 위치에 변수를 이용한다면 위의 함수는 다음과 같이 동등하게 작성할 수 있다.

```
# let f = fun a b ->
    let (x, y) = a in
    let (v, w) = b in
    x * v + y * w;;
val f : int * int -> int * int -> int = <fun>
```

위의 함수 타입은 (int * int) -> (int * int) -> int와 동등하다.

3 유닛과 화면출력

튜플은 순서쌍의 일반화된 형태이므로 보통 2개 이상의 연산식을 포함한다. 특수한 경우로 튜플이 연산식을 하나만 포함한 경우를 생각할 수 있다.

(⟨*연산식*⟩)

위의 연산식은 튜플로 해석되지 않으며 연산자 우선순위 등을 바꾸기 위해서 ⟨*연산식*⟩을 괄호로 묶은 것으로 해석된다. 예를 들어 (3 + 5) * 2에서 (3 + 5)는 튜플이 아니라 3 + 5로 해석된다.

또 다른 튜플의 특수한 경우로 괄호 안에 연산식이 아예 포함되지 않은 경우가 있다.

```
# ();;
- : unit = ()
```

연산식 ()는 유닛unit이라고 부르며 타입 unit을 가진다.

유닛은 그 자체로 계산에 필요한 값을 전달하는 것이 아니며 보통 구체적인 결과가 필요 없는 계산에서 결과값으로 넘길 때 이용한다. 예를 들어 기본적으로 제공되는 함수인 print_int는 주어진 정수를 화면에 출력하며 구체적인 계산 결과를 넘기지 않으므로 결과값으로 유닛을 이용한다.

```
# print_int;;
- : int -> unit = <fun>
# print_int (1 + 1);;
2- : unit = ()
```

맨 마지막 줄에서 2는 print_int 함수가 화면에 출력한 숫자 2이며, unit은 결과값의 타입, ()는 결과값이다.

화면출력에 이용할 수 있는 함수로는 print_int외에 다음과 같은 함수가 있다. 이 함수들은 모두 기본적으로 제공된다.

- `print_string`은 `string -> unit` 타입을 가지며 주어진 문자열을 화면에 출력한다.
- `print_float`는 `float -> unit` 타입을 가지면 주어진 부동소수를 화면에 출력한다.
- `print_newline`은 `unit -> unit` 타입을 가지며 출력 위치를 다음 줄 처음으로 바꾼다. 이 함수는 특별한 인자를 필요로 하지 않기 때문에 `unit` 타입의 인자를 받는다.

다음은 위의 함수를 이용해서 작성한 간단한 정수 비교 함수이다. 이 함수는 주어진 정수 a와 b의 대소비교 결과를 화면에 출력하고 유닛을 넘긴다.

```
let cmp a b =
  let _ = print_string "integer a = " in
  let _ = print_int a in
  let _ = print_newline () in
  let _ = print_string "integer b = " in
  let _ = print_int b in
  let _ = print_newline () in
  let msg =
    if a > b then "a is greater than b"
    else if b > a then "b is greater than a"
    else "a and b are equal"
    in
  let _ = print_string msg in
  let _ = print_newline () in
  ()
  ;;
```

5 장

자기호출함수

지금까지 배운 연산식은 모두 순차적인 계산을 표현한다. 조건연산식을 이용하면 분기점이 있는 계산을 표현할 수 있고 함수를 이용하면 자주 이용되는 복잡한 계산을 간략하게 표현할 수 있지만 모든 계산의 진행은 결국 처음부터 끝까지 순차적으로 진행된다. 이번 강의에서는 같은 형태의 계산을 반복하는 복잡한 계산을 간단하게 표현하는 방법을 배운다.

1 자기호출함수 정의

정수 1부터 10까지 더하는 계산을 생각해보자. 정수 덧셈 연산자를 여러번 이용하는 연산식으로 이 계산을 쉽게 표현할 수 있다.

```
# 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;;
- : int = 55
```

그러나 위의 방식은 일반적인 해답이 될 수 없다. 예를 들어서 1부터 인자로 주어진 정수 n까지 더하는 함수 `sum`은 n의 값을 미리 알 수 없기 때문에 위의 방식으로는 작성할 수 없다.

```
let sum n = 1 + 2 + ... + n;;     (* ??? *)
```

제대로 작성된 함수 `sum`은 정수를 1부터 n까지 차례로 증가시키면서 하나씩 더해가는 계산을 반복해서 수행해야 한다. 이런 계산을 가능하게 하는 것이 자기호출함수[recursive function]이다.

자기호출함수의 원리는 수학에서 다루는 점화식[recurrence relation]과 같다. 예로서 다음의 점화식을 생각해보자.

- 일반항: $n > 1$인 경우 $a_n = a_{n-1} + n$.
- 초항: $a_1 = 1$.

즉 a_n의 값은 n이 1보다 크면 a_{n-1}의 계산 값을 이용하고, n가 1이면 a_{n-1}의 계산 값을 계산할 필요 없이 1을 결과값으로 가진다. 위의 점화식을 해석하면 정수 1부터 n까지의 합을 계산하는 수

열sequence a_n을 정의함을 알 수 있다. 정수 n이 주어졌을 때 a_n을 계산하는 자기호출함수도 위의 점화식과 똑같은 원리로 작성할 수 있다.

위의 점화식을 다음과 같은 사양specification의 함수 sum으로 변환해 보자.

- sum의 타입은 `int -> int`이다.
- sum의 인자 n은 $n \geq 1$을 만족한다고 가정한다.
- 함수적용 `sum` n은 a_n을 결과값으로 넘긴다.

연산식으로 작성하면 우선 다음과 같이 시작할 수 있다.

```
let sum n = ...
```

$a_{\texttt{n}}$을 계산하는 sum의 몸통 연산식은 인자 n이 1보다 큰지 같은지에 따라서 계산 방식이 달라지므로 먼저 n이 1보다 큰지 같은지를 비교해야한다.

```
let sum n =
    if n > 1 then ...
    else ...
```

만약 `n > 1`이 참이라면 위의 점화식에 따라서 $a_{\texttt{n-1}} + 1$을 계산해야한다. sum의 사양에 따르면 $a_{\texttt{n-1}}$은 `sum (n - 1)`로 계산할 수 있으므로 $a_{\texttt{n-1}} + 1$은 `sum (n - 1) + 1`로 계산할 수 있다.

```
let sum n =
    if n > 1 then sum (n - 1) + n
    else ...
```

`n > 1`이 거짓이라면 `n >= 1`이 가정으로 주어지므로 `n = 1`이 참이 된다. 이 경우 결과값은 점화식의 초항에 따라서 1이 된다.

```
let sum n =
    if n > 1 then sum (n - 1) + n
    else 1
```

위의 정의를 보면 함수 sum은 몸통 연산식에서 또 다시 자신을 이용한다. 이런 의미에서 sum은 자기호출함수라고 부르며, sum이 자기호출함수임을 명시하기 위해서 키워드 rec을 함수 정의에 적어 주어야한다.

```
let rec sum n =
    if n > 1 then sum (n - 1) + n
    else 1
```

만약 `n > 1`이 아닌 `n = 1`을 조건식으로 이용하면 sum은 다음과 같이 정의된다.

```
let rec sum n =
    if n = 1 then 1
    else sum (n - 1) + n
```

함수 sum이 위와 같이 정의되면 sum n은 $a_n = \sum_{i=1}^{n} i$를 계산함을 알 수 있다. 예를 들어 sum 10은 계산은 다음과 같이 진행된다.

```
  sum 10
↦ if 10 = 1 then 1 else sum (10 - 1) + 10
↦ if false then 1 else sum (10 - 1) + 10
↦ sum (10 - 1) + 10
↦ sum 9 + 10
↦ (if 9 = 1 then 1 else sum (9 - 1) + 9) + 10
↦ (if false then 1 else sum (9 - 1) + 9) + 10
↦ (sum (9 - 1) + 9) + 10
↦ sum 8 + 9 + 10
↦ ···
↦ sum 1 + 2 + ··· + 9 + 10
↦ (if 1 = 1 then 1 else sum (1 - 1) + 1) + 2 + ··· + 9 + 10
↦ (if true then 1 else sum (1 - 1) + 1) + 2 + ··· + 9 + 10
↦ 1 + 2 + ··· + 9 + 10
↦ ···
↦ 55
```

일반적으로 자기호출함수 f의 몸통 연산식은 f 자신의 적용이 일어나는 자기호출연산식recursive case, f의 적용이 일어나지 않는 종결연산식base case, f의 적용여부를 결정하는 종결조건termination condition을 포함하며 다음과 같은 구조를 가진다.

- if ⟨*종결조건*⟩ then ⟨*종결연산식*⟩ else ⟨*자기호출연산식*⟩
  해설: ⟨*종결조건*⟩이 만족되면 ⟨*종결연산식*⟩을 계산하고 만족되지 않으면 ⟨*자기호출연산식*⟩을 계산한다.
  예: `if n = 1 then 1 else sum (n - 1) + 1`

- if not ⟨*종결조건*⟩ then ⟨*자기호출연산식*⟩ else ⟨*종결연산식*⟩
  해설: ⟨*종결조건*⟩이 만족되지 않으면 ⟨*자기호출연산식*⟩을 계산하고 만족되면 ⟨*종결연산식*⟩을 계산한다.
  예: `if n > 1 then sum (n - 1) + 1 else 1`

자기호출함수를 정의할 때는 다음의 두 가지를 주의하여야 한다. 첫째, 종결조건의 검사가 반드시 필요하다. 종결조건을 검사하지 않는 함수는 초항이 없는 점화식을 계산하는 것과 같으며, 매번 또다시 자신을 호출하게 되므로 계산이 종결되지 않는다. 이런 경우 계산이 무한반복infinite loop에 빠진다고 말한다. 예를 들어서 위의 sum 함수를 종결조건 검사 없이 작성했다고 가정하자.

```
let rec sum n =
    sum (n - 1) + n
```

함수 sum은 실제인자의 값에 상관없이 항상 자신을 호출하므로 무한반복에 빠지게 된다. 둘째, 종결연산식에서 자신을 호출할 때 이용하는 인자는 값이 변해야 한다. 인자 값이 변하지 않는 함수는 점화식의 일반항에서 a_n을 또다시 a_n을 이용해서 적는 것과 같으며, 종결조건의 검사 여부에 상관없이 같은 계산을 반복하게 되어 역시 무한반복에 빠진다. 예를 들어서 위의 sum 함수를 다음과 같이 잘못 작성했다고 가정하자.

```
let rec sum n =
     if n = 1 then 1
     else sum n + n
```

이 경우 sum n은 n이 1이 아니면 sum n을 다시 계산하므로 무한반복에 빠진다.

자기호출함수를 적용할 때 인자에 대한 가정이 만족되는지 확인해야 한다. 예를 들어 다음과 같은 sum의 정의를 생각해보자.

```
let rec sum n =
     if n = 1 then 1
     else sum (n - 1) + n
```

이 함수는 인자 n이 1보다 크거나 같다고 가정하고 작성되었으며 0보다 작거나 같은 정수에 잘못 적용이 되면 무한반복이 발생한다. 즉 자기호출함수는 연산식 작성뿐만 아니라 연산식에 이용할 때도 무한반복이 발생하지 않도록 주의를 해야 한다.

자기호출함수를 잘못 작성하거나 잘못 적용하여 무한반복이 발생하면 보통 예외상황의 발생으로 이어지고 계산의 결과 없이 끝나게 된다. 다음 예는 위의 잘못 작성된 sum 함수를 -10에 적용시켰을 때의 결과이다.

```
# sum (-10);;
Stack overflow during evaluation (looping recursion?).
```

2 자기호출함수의 예

자기호출함수는 sum 함수와 같이 하나의 인자를 가지고 몸통 연산식에서 자신을 한번만 호출할 필요는 없다. 좀 더 일반적인 형태의 자기호출함수를 살펴보자.

2.1 자신을 여러번 이용하는 자기호출함수

피보나치Fibonacci 수열 F_n은 다음과 같은 점화식으로 정의된다.

- 일반항: $n > 2$인 경우 $F_n = F_{n-1} + F_{n-2}$.

- 초항: $F_2 = F_1 = 1$.

위의 점화식을 다음과 같은 사양의 함수 `fib`으로 변환해 보자.

- `fib`의 타입은 `int -> int`이다.
- `fib`의 인자 n은 $n \geq 1$을 만족한다고 가정한다.
- 함수적용 `fib` n은 F_n을 결과값으로 넘긴다.

`fib`는 다음과 같이 작성할 수 있다.

```
let rec fib n =
    if n > 2 then fib (n - 1) + fib (n - 2)
    else 1
```

함수적용 `fib` n은 $n > 2$일 때 `fib`를 두번 호출한다. 즉 F_{n-1}을 계산하기 위해서 한번, 그리고 F_{n-2}를 계산하기 위해서 또 한번 `fib`를 호출한다. `fib`는 자신을 두번 호출하지만 인자를 1 또는 2 감소시키므로 결국은 종결조건을 만족하는 인자에 도달하게 된다. 따라서 계산과정은 함수 `sum`의 경우보다 복잡하지만 무한반복은 발생하지 않는다.

2.2 값이 변하지 않는 인자를 여러 개 가지는 자기호출함수

k라는 상수가 주어졌을 때 다음과 같은 점화식을 생각해보자.

- 일반항: $n > 1$인 경우 $a_n = ka_{n-1} + 1$.
- 초항: $a_1 = 1$.

위의 점화식을 다음과 같은 사양의 함수 `f`로 변환해 보자.

- `f`의 타입은 `int -> int`이다.
- `f`의 인자 n은 $n \geq 1$을 만족한다고 가정한다.
- 함수적용 `f` n은 a_n을 결과값으로 넘긴다.

점화식에서 이용하는 상수 k가 변수 `k`에 저장되어 있다면 함수 `f`는 다음과 같이 정의할 수 있다.

```
let rec f n =
    if n > 1 then k * (f (n - 1)) + 1
    else 1
```

위의 자기호출함수는 수열 a_n을 올바르게 계산하지만 고정된 값 k에 대해서만 이용할 수 있다. 예를 들어 위의 함수가 정의되는 시점에서 변수 `k`에 10이라는 값이 저장되어 있다고 하자. 그러면 위의 함수는 점화식의 일반항이 $a_n = 10a_{n-1} + 1$인 수열을 계산한다. 그러나 나중에 점화식의 일

반항이 $a_n = 20a_{n-1} + 1$로 바뀐 수열을 계산하고자 한다면 위의 함수 f는 더 이상 이용할 수 없고 새로운 자기호출함수를 다시 정의해야한다.

점화식의 상수 k를 고정시키지 않고 언제든지 바꿀 수 있게 하려면 다음과 같은 사양의 함수 f_gen을 작성하면 된다.

- f_gen의 타입은 `int -> int -> int`이다.
- f_gen의 두번째 인자 n은 $n \geq 1$을 만족한다고 가정한다.
- 함수적용 `f_gen k n`은 a_n을 결과값으로 넘긴다.

f_gen은 결과값을 넘기기 위해서 두개의 정수를 인자로 필요로 하므로 다음과 같이 몸통 연산식에서도 두 개의 인자를 전달하여 함수 f_gen을 호출하여야 한다.

```
let rec f_gen k n =
    if n > 1 then k * (f_gen k (n - 1)) + 1
    else 1
```

여기서 첫번째 인자 k는 점화식의 형태만을 결정하므로 몸통 연산식에서 f_gen을 호출할 때에도 같은 값을 전달한다. 반면에 두번째 인자 n은 수열내의 순서를 결정하므로 몸통 연산식에서 f_gen을 호출할 때 1 감소하여 전달된다. 이 함수를 이용하면 일반항이 $a_n = 10a_{n-1} + 1$인 수열의 n번째 값은 `f_gen 10 n`으로, 일반항이 $a_n = 20a_{n-1} + 1$인 수열의 n번째 값은 `f_gen 20 n`으로 계산할 수 있다. 즉 함수 f_gen 하나만 정의하여 여러 k값에 대해서 수열 a_n을 계산할 수 있다.

함수 f_gen을 더 일반화시키면 다음과 같은 점화식을 계산하는 함수 f_full을 작성할 수 있다.

- 일반항: $n > 1$인 경우 $a_n = ka_{n-1} + p$.
- 초항: $a_1 = q$.

```
let rec f_full k p q n =
        if n > 1 then k * (f_full k p q (n - 1)) + p
        else q
```

f_full의 타입은 `int -> int -> int -> int -> int`가 된다.

2.3 여러 개의 값을 계산하는 자기호출함수

지금까지 살펴본 자기호출함수는 모두 결과값으로 하나의 정수를 계산한다. 그러나 튜플을 이용하면 여러 개의 값을 동시에 계산하는 자기호출함수를 작성할 수 있다. 예로서 두 수열 a_n과 b_n을 동시에 정의하는 다음 점화식을 생각해보자.

- 일반항: $a_n = 2a_{n-1} + b_{n-1}$
  $b_n = a_{n-1} + 2b_{n-1}$

- 초항: $a_1 = p$
  $b_1 = q$

수열 a_n과 b_n을 계산하는 함수는 입력으로 n을 받아서 a_n과 b_n으로 이루어진 튜플을 생성하게 된다. 따라서 위의 점화식은 다음과 같은 사양의 함수 `f_pair`로 변환할 수 있다.

- `f_pair`의 타입은 `int -> int * int`이다. (이 타입은 `int -> (int * int)`와 같다.)
- `f_pair`의 인자 n은 $n \geq 1$을 만족한다고 가정한다.
- 함수적용 `f_pair` n은 `(`a_n`,` b_n`)`을 결과값으로 넘긴다.

점화식에서 이용하는 상수 p와 q가 변수 `p`와 `q`에 저장되어 있다면 `f_pair`는 패턴검사를 이용하여 다음과 같이 정의할 수 있다.

```
let rec f_pair n =
    if n > 1 then
       let (a, b) = f_pair (n - 1) in
       (2 * a + b, a + 2 * b)
    else
       (p, q)
```

여기서 `f_pair (n - 1)`의 계산결과는 두개의 정수로 이루어지며 각각 변수 `a`와 `b`에 저장된다.

2.4 값이 변하는 인자를 여러 개 가지는 자기호출함수

지금까지 살펴본 자기호출함수는 모두 값이 변하는 인자를 하나만 가진다. 그러나 값이 변하는 인자를 여러 개 가지는 자기호출함수를 생각해 볼 수도 있다. 다음의 자기호출함수 `gcd`는 두개의 `int` 타입 인자를 가지며 몸통 연산식에서 자신을 호출할 때 두개의 인자 모두 바뀔 수 있음을 보여주는 예이다.

```
let rec gcd a b =
    if a > b then gcd (a - b) b
    else if a < b then gcd a (b - a)
    else a
```

`gcd`의 몸통 연산식은 `a`와 `b`가 같은 값을 가질 때 `a`를 반환하므로 종결조건은 `a = b`가 된다. `gcd` a b는 $a > 0$와 $b > 0$이 성립할 때 a와 b의 최대공약수greatest common divisor를 유클리드호제법Euclidean algorithm을 이용하여 계산한다.

함수적용 `gcd` a b는 $a > 0$와 $b > 0$이 성립하면 무한반복에 빠지지 않는다. 만약 $a = b$가 성립하면 종결조건이 만족되어 계산은 끝이 난다. 그렇지 않으면 a와 b의 대소 관계에 따라서 `gcd` $(a - b)$ b 또는 `gcd` a $(b - a)$가 계산되는데 두 인자의 합이 $a + b$에서 a 또는 b로 줄어드는

것을 알 수 있다. 두 인자의 합은 항상 양수이므로 함수의 자기 호출은 언젠가는 끝이 나게 되어있다. 만약 무한반복이 발생한다면 두 인자의 합이 무한번 감소하여도 양수로 남는다는 결론이 나오는데 이는 모순이기 때문이다.

3 꼬리물기 자기호출함수

지금까지 살펴본 자기호출함수는 계산의 크기를 결정하는 인자와 값이 변하지 않는 인자를 가진다. (계산의 크기를 결정하는 인자는 보통 종결조건에 이용한다.) 종결조건에 이용하는 인자는 계산의 크기를 결정하는 주된 역할을 가지므로 반드시 필요하며 몸통 연산식에서 함수적용이 일어날 때 그 값이 변한다. 값이 변하지 않는 인자는 계산의 크기와는 상관없는 보조적인 역할을 가지므로 필요없을 수도 있다. 예를 들어 2.1절의 함수 `fib`은 계산의 크기를 결정하는 인자 n 하나만 이용하며 2.4절의 함수 `f_full`은 인자로 계산의 크기를 결정하는 인자 n 외에 값이 변하지 않는 인자 k, p, q를 이용한다. 꼬리물기 자기호출함수[tail-recursive function] 또는 꼬리물기함수는 자기호출함수의 새로운 형태로서 계산의 중간결과를 저장하는 인자를 가진다.

꼬리물기함수의 작동원리를 설명하기 위해서 1부터 주어진 정수까지 더하는 자기호출함수 `sum`을 생각해보자. (`sum`은 꼬리물기함수가 아니다.)

```
let rec sum n =
    if n > 1 then n + sum (n - 1)
    else 1
```

n이 1보다 큰 경우 연산식 `sum` n의 계산은 `sum (`n `- 1)`의 계산결과를 필요로 한다. 따라서 `sum (`n `- 1)`의 계산을 마칠 때까지 `sum` n의 계산도 대기하고 있어야 한다. `sum (`n `- 1)`의 계산이 종결되어 결과값 s가 넘어오고 나서야 `sum` n은 $n+s$을 결과값으로 넘길 수 있게 된다.

`sum 10`의 계산을 예로 들어보자. `sum 10`은 `sum 9`의 계산결과를 기다리고, `sum 9`는 `sum 8`의 계산결과를 기다리고, 이 과정이 반복되어 `sum 2`의 계산이 `sum 1`의 계산결과를 기다릴 때까지 진행된다. `sum 1`이 `1`을 반환하면 `sum 2`는 `3`을 반환하며, 이 과정이 반복되어 `sum 9`는 `45`를 반환하고 최종적으로 `sum 10`은 `55`를 반환하게 된다.

```
sum 10  →  sum 9  →  sum 8  →···→  sum 2  →  sum 1
                                                ↓
10 + 45  ←  9 + 36  ←  8 + 48  ←···←  2 + 1  ←    1
```

따라서 `sum 1`의 계산이 진행되는 동안 함수적용 `sum 10`, `sum 9`, `sum 8`, ···, `sum 2`의 계산은 모두 대기 상태가 된다. 만약 `sum 1000000000`과 같이 매우 큰 수를 `sum`의 인자로 주면 대기 상태의 함수적용의 개수가 늘어나고 컴퓨터의 용량의 한계로 보통 스택넘침현상[stack overflow]이라는 예외상황으로 이어진다.

```
# sum 100000;;
Stack overflow during evaluation (looping recursion?).
```

그러면 어떻게 하면 함수적용을 대기상태로 만들 필요가 없어질까? 즉 어떻게 하면 다음 그림과 같이 `sum 10`의 계산은 `sum 9`의 계산으로 변형되어 `sum 9`의 계산결과를 기다릴 필요가 없고, `sum 9`의 계산은 `sum 8`의 계산으로 변형되어 `sum 8`의 계산결과를 기다릴 필요가 없게 될까?

```
sum 10  →  sum 9  →  sum 8  →···→  sum 2  →  sum 1  →  ???
```

이를 해결하는 방법은 함수 적용이 일어날 때 기존의 인자 이외에 지금까지 더해진 정수의 합을 함께 넘기는 것이다. 이렇게 두 개의 인자를 받는 함수를 `sum'`이라고 하면 기존의 `sum 10`의 계산은 다음과 같이 완료할 수 있다.

```
  sum 10
    ↓
sum' 10 0  →  sum' 9 10  →  sum' 8 19  →···→  sum' 2 52  →  sum' 1 54
                                                              ↓
                                                              55
```

즉 먼저 `sum 10`의 계산을 `sum' 10 0`의 계산으로 변환한다. 여기서 `sum'`의 첫번째 인자 `10`은 앞으로 더해갈 정수들 중 첫번째이며 두번째 인자 `0`은 지금까지 더한 정수의 합이 된다. 위의 계산이 진행되는 방식은 다음과 같다.

- 지금까지 더한 정수의 합은 0이고 앞으로 10부터 더해가는 계산은 지금까지 더한 정수의 합을 10으로 두고 앞으로 9부터 더해가는 계산과 동등하다.
- 지금까지 더한 정수의 합은 10이고 앞으로 9부터 더해가는 계산은 지금까지 더한 정수의 합을 19로 두고 앞으로 8부터 더해가는 계산과 동등하다.
- ···
- 지금까지 더한 정수의 합이 54이고 앞으로 1부터 더해가야 하는 계산은 55를 반환함으로써 바로 종결할 수 있다.

여기서 중요한 점은 `sum'` n s를 `sum'` $(n-1)$ $(n+s)$로 변환하면 `sum'` n s는 `sum'` $(n-1)$ $(n+s)$의 계산결과를 기다릴 필요가 없다는 것이다. 왜냐하면 최종 계산결과는 `sum' 1 54`에서 `55`로 계산되기 때문이다. (함수 `sum`의 경우는 최종 계산결과는 `sum 1`에서 계산되지 않고 다시 `sum 10`에서 계산되었다.) 이렇게 계산의 중간결과를 인자로 전달하여 몸통 연산식에서 자신을 호출하고 난 뒤 계산결과를 기다릴 필요가 없도록 정의된 함수가 꼬리물기함수이다.

위의 `sum'`을 이용한 꼬리물기함수는 다음과 같이 정의할 수 있다.

```
let sum n =
  let rec sum' i s =
    if i > 1 then sum' (i - 1) (s + i)
    else s + 1
    in
  sum' n 0
```

즉 함수 sum은 꼬리물기함수 sum'을 지역변수로 정의한 뒤 sum n의 계산을 sum' n 0로 변환한다. 여기서 sum 자체는 자기호출함수로 정의되지 않는다. 이렇게 정의된 함수 sum은 스택넘침현상이 발생하지 않는다.

```
# sum 100000;;
- : int = 705082704
```

(여기서 705082704는 정수의 넘침현상 때문에 발생한 부정확한 계산결과이다.)

자기호출함수가 꼬리물기함수가 되기 위해서는 반드시 몸통 연산식에서 계산의 마지막 부분이 자신을 호출하는 함수적용이 되어야 한다. 그렇지 않으면 함수적용의 계산이 완료될 때까지 몸통 연산식의 계산은 대기하고 있어야 하는 상황이 발생하므로 꼬리물기함수가 되지 않는다. 예를 들어 위의 sum'을 다음과 같이 sum''으로 재정의하면 비록 함수의 인자로 중간계산결과가 전달된다 하더라도 sum'' (i - 1) (s + i)의 계산결과를 1과 더해야하므로 꼬리물기함수가 되지 않는다.

```
let rec sum'' i s =
  if i > 1 then (sum'' (i - 1) (s + i)) + 1
  else s + 1
```

꼬리물기함수에서 중간계산결과를 저장하는 인자의 값은 계산의 크기를 결정하는 인자와 올바르게 쌍을 이루도록 해야 한다. 예를 들어 sum n 내에서 정의되는 꼬리물기함수 sum'은 모든 함수적용 sum' i s에 대해서 $s = \sum_{j=i+1}^{n} j$의 조건이 성립하도록 정의되어있다. sum의 몸통 연산식에 있는 함수적용 sum' n 0과 sum' (i - 1) (s + i)가 실제로 이 조건을 만족하고 있음은 다음과 같이 확인할 수 있다.

- sum' n 0에서 $0 = \sum_{j=n+1}^{n} j$가 성립한다.
- sum' i s에서 $s = \sum_{j=i+1}^{n} j$가 성립한다고 가정하면 sum' $(i-1)$ $(s+i)$에서 $s + i = \sum_{j=(i-1)+1}^{n} j$가 성립한다.

따라서 sum' 1 s가 계산될 때 $s = \sum_{j=1+1}^{n} j$를 만족하므로 $s + 1$은 원하는 결과인 $\sum_{j=1}^{n} j$를 최종적으로 계산하게 된다.

자기호출함수를 꼬리물기함수로 변형하는 방식은 정해져 있지 않다. 즉 같은 자기호출함수를 여러 다른 형태의 꼬리물기함수로 변형할 수 있다. (모든 자기호출함수는 꼬리물기함수로 변형할 수 있음은 이미 증명되어 있다.) 예를 들어 자기호출함수 sum은 다음과 같이 1부터 정수를 더해가는 꼬리물기함수로 변형할 수도 있다.

```
let sum n =
    let rec sum_reverse i s =
      if i = n then s
      else sum_reverse (i + 1) (s + i + 1)
      in
    sum_reverse 1 1
```

여기서 꼬리물기함수 `sum_reverse`는 함수적용 `sum_reverse` i s가 있을 때 $s = \sum_{j=1}^{i} j$의 조건을 만족하도록 정의되어 있다.

꼬리물기 자기호출함수도 일반적인 자기호출함수처럼 값이 변하지 않는 인자를 여러개 가질 수도 있고 (2.2절 참조), 여러개의 값을 계산할 수도 있을 뿐 아니라 (2.3절 참조) 여러개의 인자에 중간계산결과를 저장할 수도 있다. (그러나 몸통연산식에서 자신을 여러번 이용하는 꼬리물기함수는 있을 수가 없다!) 다음은 2.1절의 피보나치 수열을 계산하는 자기호출함수를 꼬리물기함수로 변형한 예이다. 이 함수는 중간계산결과를 두 개의 인자에 저장한다.

```
let fib n =
  let rec fib' i p q =
    if i = n then p
    else fib' (i + 1) (p + q) p
  in
  if n = 1 then 1
  else fib' 2 1 1
```

함수 `fib` 내에 정의되어 있는 꼬리물기함수 `fib'`은 모든 함수적용 `fib` i p q에 대해서 $p = F_i$ 그리고 $q = F_{i-1}$의 조건을 모두 만족하도록 정의되어 있다.

4 상호호출함수

지금까지 살펴본 자기호출함수는 몸통 연산식에서 자신만을 호출할 수 있다. 그러나 두개의 함수가 서로 상대방을 호출하는 것도 가능하며 이렇게 정의되는 함수를 상호호출함수mutually recursive function이라고 한다.

간단한 예로서 주어진 0 이상의 정수가 짝수even number인지를 판별하는 함수 `even`과 홀수odd number인지 판별하는 함수 `odd`를 작성해보자.

- 함수적용 `even` n은 n이 짝수이면 불린형값 `true`를, 아니면 `false`를 결과값으로 넘긴다.
- 함수적용 `odd` n은 n이 홀수이면 불린형값 `true`를, 아니면 `false`를 결과값으로 넘긴다.

함수 `even`과 `odd`는 각각 별도의 자기호출함수로 작성할 수 있다. (`odd`는 유사하게 작성할 수 있으므로 생략한다.)

```
let rec even n =
  if n = 0 then true
  else if n = 1 then false
  else even (n - 2)
```

그러나 n이 0보다 클 때 n이 짝수이면 $n-1$이 홀수이고, n이 홀수이면 $n-1$이 짝수라는 사실을 이용하면 even과 odd는 다음과 같이 동시에 작성할 수 있다.

```
let rec even n =
  if n = 0 then true
  else odd (n - 1)
and odd n =
  if n = 0 then false
  else even (n - 1)
```

즉 함수 even과 odd는 인자가 0인 경우 바로 true나 false를 결과값으로 반환하지만 그렇지 않은 경우에는 인자를 1 감소시킨 뒤 상대방을 호출하도록 정의되어 있다. 즉 상호호출함수로 정의되고 있다. 여기서 키워드 and는 함수 even과 odd가 동시에 정의되고 있음을 나타낸다. 만약 and를 제거하고 let rec으로 바꾸면 even의 몸통 연산식에서 아직 정의가 되지 않은 함수 odd를 이용하므로 오류가 발생한다. even과 odd는 몸통 연산식 계산의 마지막 부분이 함수적용이므로 꼬리물기함수로 간주할 수 있다. (따라서 인자가 커도 스택넘침현상이 발생하지 않는다.)

6 장

고차함수

지금까지 정수나 부동소수와 같은 기본적인 자료형의 조합을 인자로 받고 결과값으로 반환하는 함수에 대해서 배웠다. 이번 강의에서는 함수 자체를 다른 함수의 인자로 이용하거나 결과값으로 이용하는 방법을 배운다.

1 고차함수의 의미

계산은 무엇을 어떻게 처리하여 결과값을 얻는지 설명하는 것으로 이루어진다. 여기서 '무엇'과 '결과값'은 계산의 대상체로서 정수나 부동소수와 같은 기본자료형의 조합으로 표현하며, '어떻게'는 계산의 중간과정으로서 함수를 이용하여 표현한다. 사실 기본적으로 제공되는 +와 *와 같은 연산자도 내부적으로는 함수로 정의되어 있기 때문에 모든 계산의 중간과정이 함수로 표현된다는 것은 틀린 설명이 아니다. 이렇게 함수는 계산에서 대상체를 어떻게 변환하고 생성하는지를 표현하는 특별한 지위를 가지며, 정수나 부동소수와 같은 기본자료형은 계산의 대상체로만 이용되는 보조적인 지위를 가진다.

정수와 부동소수와 같이 함수의 인자나 결과값으로 이용할 수 있는 자료형을 일종객체first-class object라고 부른다. 여기서 '일종'이란 특별한 지위를 가진다는 뜻에서의 일종이 아니라 (특별한 처리를 요하는 특급우편물과는 대조되는) 편지나 엽서와 같은 일종우편물처럼 일반적인 처리를 요한다는 뜻에서의 일종이다. 즉 정수나 부동소수는 함수 간에 전달되는 일종우편물에 비유하는 것이다.

그러면 함수 자체를 일종객체로 이용할 수 있을까? 즉 함수의 인자로서 함수를 전달받고 함수의 결과값으로 함수를 반환할 수 있을까? 대부분의 프로그래밍언어는 함수를 일종객체로 허용하지 않지만 OCAML의 경우는 허용한다. 따라서 함수도 일종객체의 위치를 가지게 되고 이러한 의미에서 OCAML에서 정의하는 모든 함수는 일종함수first-class function라고 말한다.

함수를 인자나 결과값으로 이용하는 함수를 고차함수higher-order function이라고 한다. (기본자료형만을 인자와 결과값으로 이용하는 함수는 일차함수first-order function이라고 한다. 지금까지 다룬 모든 함수는 일차함수이다.) 고차함수는 다음과 같은 세 종류로 나눌 수 있다. 가장 일반적인 형태의 고차함수는 세번째이다.

- 함수를 인자로 받고 기본자료형만을 결과값에 이용하는 고차함수

- 기본자료형만으로 이루어진 인자를 받아서 함수를 결과값으로 반환하는 고차함수
- 기본자료형과 함수를 인자와 결과값에 모두 이용하는 고차함수

다음 절에서는 여러 가지 예를 통해서 고차함수가 어떤 경우에 유용한지를 설명한다.

2 고차함수의 예

3장에서 대상체만 바뀌고 중간과정은 동일한 계산이 반복될 때 함수를 이용하면 전체 연산식을 간단하게 표현할 수 있음을 배웠다. 마찬가지로 이용하는 함수만 바뀌고 중간과정은 동일한 계산이 반복될 때 고차함수를 이용하면 전체연산식을 간단하게 표현할 수 있다. 예를 통해서 고차함수가 실제로 어떻게 이용되는지 살펴보자.

2.1 함수를 인자로 받는 고차함수

함수 f를 받아서 0에 적용한 결과를 반환하는 고차함수를 생각해보자. 모든 함수는 일종객체이므로 정수나 부동소수형 인자를 이용할 때와 같은 방식으로 f를 형식인자로 이용하면 된다.

```
let app_zero f = f 0
```

몸통 연산식에서 `f`는 정수 연산식 `0`에 적용되므로 `int` 타입을 인자로 받는 모든 함수에 `app_zero`를 적용할 수 있다. 예를 들어 주어진 정수를 1 증가시키거나 감소시키는 함수 `inc`와 `dec`를 가정하자.

```
let inc = fun x -> x + 1;;
let dec = fun x -> x - 1;;
```

`app_zero`를 `inc`과 `dec`에 적용하면 각각 `inc 0`과 `dec 0`을 계산한다.

```
# app_zero inc;;
- : int = 1
# app_zero dec;;
- : int = -1
```

조금 더 복잡한 예로서 함수 f를 받아서 0에 두번 적용한 결과를 반환하는 고차함수를 정의해 보자.

```
# let app_twice_zero f = f (f 0);;
val app_twice_zero : (int -> int) -> int = <fun>
```

`app_twice_zero`의 타입 `(int -> int) -> int`를 그대로 해석하면 `int -> int` 타입의 값을 받아서 `int` 타입의 값을 반환함을 의미한다. 그런데 `int -> int` 타입의 값은 사실 `int -> int` 타입을 가지는 함수이다. 따라서 `app_twice_zero`의 타입은 `int -> int` 타입의 함수를 받아서 정수를 반환함을 나타낸다. `app_twice_zero`를 `inc`과 `dec`에 적용하면 각각 `inc (inc 0)`과 `dec (dec 0)`을 계산한다.

```
# app_twice_zero inc;;
- : int = 2
# app_twice_zero dec;;
- : int = -2
```

함수 f를 받아서 0에 세번 적용한 결과를 반환하는 고차함수도 유사하게 정의할 수 있다.

```
# let app_thrice_zero f = f (f (f 0));;
val app_thrice_zero : (int -> int) -> int = <fun>
# app_thrice_zero inc;;
- : int = 3
# app_thrice_zero dec;;
- : int = -3
```

위의 함수들을 일반화하면 함수 f를 받아서 0에 n번 적용한 결과를 반환하는 고차함수 app_times_zero를 자기호출함수 형태로 정의할 수 있다.

```
# let rec app_times_zero f n =
  if n = 0 then 0
  else f (app_times_zero f (n - 1));;
val app_times_zero : (int -> int) -> int -> int = <fun>
```

만약 $n = 0$이면 f는 0번 적용되므로 (즉 한번도 적용되지 않으므로) 결과값은 0이 된다. 그렇지 않으면 f를 0에 $(n-1)$번 적용한 결과에 다시 한번 더 f를 적용하여 총 n번을 적용하게 된다. app_times_zero는 다음과 같이 이용할 수 있다.

```
# app_times_zero inc 10;;
- : int = 10
# app_times_zero dec 10;;
- : int = -10
```

2.2 함수를 결과값으로 반환하는 고차함수

함수를 결과값으로 반환하기 위해서는 fun 키워드를 이용하여 만든 연산식이 함수를 나타내는 하나의 값이라는 점을 이해하면 된다. 예를 들어 inc의 정의를 살펴보자.

```
let inc = fun x -> x + 1
```

이 정의를 그대로 해석하면 연산식 fun x -> x + 1의 계산결과가 변수 inc에 저장됨을 알 수 있다. 그런데 fun x -> x + 1은 그 자체가 이미 하나의 값으로서 주어진 정수를 1 증가시키는 함수를 나타낸다. 따라서 이 정의는 실질적으로 주어진 정수를 1 증가시키는 함수 inc를 생성하는 것으로 해석되는 것이다.

fun 키워드를 이용하여 함수를 생성할 때 다른 지역변수나 인자를 이용할 수 있다. 예를 들면 다음 함수는 인자로 정수 n이 주어졌을 때 주어진 정수를 n 증가시키는 새로운 함수를 반환한다.

```
# let incr_n n = fun x -> x + n;;
val incr_n : int -> int -> int = <fun>
```

따라서 incr_n 10은 주어진 정수를 10 증가시키는 함수가 되고 incr_n (-10)은 주어진 정수를 10 감소시키는 함수가 된다.

```
# (incr_n 10) 0;;
- : int = 10
# (incr_n (-10)) 0;;
- : int = -10
```

함수 incr_n은 정수를 받아서 int -> int 타입의 함수를 반환하므로 int -> (int -> int) 타입을 가지게 된다. 그런데 위의 실행 예에서는 타입이 int -> int -> int로 출력되고 있는데 이는 int -> int -> int와 int -> (int -> int)는 완전히 동등한 타입임을 의미한다. 설명을 돕기 위해서 incr_n과 비슷한 함수 incr_n'을 생각해보자.

```
# let incr_n' n x = x + n;;
val incr_n' : int -> int -> int = <fun>
```

incr_n'은 정수 n과 x를 인자로 받아서 $x + n$을 결과값으로 반환하므로 int -> int -> int 타입을 가진다. 중요한 점은 이 두개의 함수 incr_n과 incr_n'이 완전히 동등하다는 점이다. 즉 다음과 같이 incr_n을 incr_n'으로 해석해도 되고 반대로 incr_n'을 incr_n으로 해석해도 된다. (3장에서 소개한 키워드 fun을 이용하지 않는 함수 정의방식도 같은 원리를 이용하고 있다.)

- incr_n은 incr_n'과 동등하므로 incr_n 10 0의 계산결과는 10이다.
- incr_n'은 incr_n과 동등하므로 incr_n' 10의 계산결과는 주어진 정수를 10 증가시키는 함수가 된다.

```
# incr_n 10 0;;
- : int = 10
# incr_n' 10;;
- : int -> int = <fun>
```

이 원리를 이용하면 함수를 결과값으로 반환하는 고차함수를 여러 가지 형태로 작성할 수 있다. 예를 들면 정수 a와 b를 인자로 받아서 함수 $f(x) = ax + b$를 반환하는 고차함수 linear는 다음과 같이 여러 가지 형태로 작성할 수 있다.

```
let linear a b x = a * x + b;;
let linear a b = fun x -> a * x + b;;
let linear a = fun b x -> a * x + b;;
let linear a = fun b -> fun x -> a * x + b;;
let linear = fun a -> fun b -> fun x -> a * x + b;;
let linear = fun a b x -> a * x + b;;
```

이 원리를 일반화하면 다음 두 타입은 완전히 동등하다는 결론을 얻게 된다.

$$T_1 \texttt{ -> } T_2 \texttt{ -> } \cdots \texttt{ -> } T_{n-1} \texttt{ -> } T_n$$
$$T_1 \texttt{ -> (} T_2 \texttt{ -> } \cdots \texttt{ -> (} T_{n-1} \texttt{ -> } T_n \texttt{)} \cdots \texttt{)}$$

2.3 함수를 인자로 받고 결과값으로 반환하는 고차함수

세번째 형태의 고차함수는 위에서 살펴본 고차함수 형태들을 일반화시킨 경우이다. 예로서 함수 f와 g를 받아서 새로운 함수 $h(x) = f(x) + g(x)$를 생성하는 고차함수 `combine`은 다음과 같이 작성할 수 있다.

```
let combine f g = let h x = f x + g x in h
```

만약 임시로 `h`라는 함수를 생성하지 않고 바로 최종 결과값을 반환하고자 하면 다음과 같이 작성하면 된다.

```
let combine f g = fun x -> f x + g x
```

이 정의는 2.2절에서 설명한 원리를 이용하여 다음과 같이 간단하게 만들 수 있다.

```
let combine f g x = f x + g x
```

다음 예는 `combine`을 이용하여 새로운 함수를 만든 뒤 0에 적용하는 예이다.

```
# let h = combine (fun x -> x + 1) (fun y -> y - 1) in h 0;;
- : int = 0
```

여기서 함수 `fun x -> x + 1`과 `fun y -> y - 1`이 정수와 부동소수처럼 값으로 취급되고 있음을 볼 수 있다.

다른 예로서 함수 f와 g를 받아서 합성composition 함수 $g \circ f$를 반환하는 고차함수 `compose`는 다음과 같이 작성할 수 있다.

```
let comp f g = fun x -> g (f x)
```

이 정의는 다음과 같이 간단하게 만들 수 있다.

```
let comp f g x = g (f x)
```

3 타입변수와 다형함수

고차함수를 작성하다 보면 함수의 타입을 정확하게 결정할 수 없는 상황이 종종 발생한다. 예를 들어 2.3절에서 정의한 `combine`을 생각해보자.

```
let combine f g = fun x -> f x + g x
```

연산식 `f x + g x`로부터 다음과 같은 두 가지 사실을 추론할 수 있다.

- 함수적용 `f x`와 `g x`는 실제인자 `x`를 공유하므로 함수 `f`와 `g`의 형식인자 타입은 동일하다.
- 함수 `f`와 `g`는 모두 `int` 타입의 값을 반환한다.

따라서 `combine`은 구체적으로 결정할 수 없는 타입 T에 대해서

$$(T \texttt{ -> int}) \texttt{ -> } (T \texttt{ -> int}) \texttt{ -> } (T \texttt{ -> int})$$

타입을 가진다고 결론을 내릴 수 있다. 여기서 중요한 점은 타입 T는 사실 아무 타입이나 될 수 있다는 사실이다. 예를 들어 T `= int`로 해석하여 `int -> int` 타입의 함수들을 인자로 이용할 수 있고 T `= bool`로 해석하여 `bool -> int` 타입의 함수들을 인자로 이용할 수 있다.

```
# combine (fun x -> x + 1) (fun y -> y - 1);;
- : int -> int = <fun>
# combine (fun x -> if x then 1 else 0) (fun y -> if y then 0 else 1);;
- : bool -> int = <fun>
```

OCAML에서는 T와 같이 구체적으로 결정할 수 없는 타입을 타입변수[type variable]라고 부르며 `'a`, `'b`, `'c` 등의 형태로 표시한다. 예를 들어 `combine`의 타입을 표시할 때 타입변수 `'a`를 이용할 수 있다.

```
# let combine f g = fun x -> f x + g x;;
val combine : ('a -> int) -> ('a -> int) -> 'a -> int = <fun>
```

보통 `'a`가 포함된 타입은 "아무 타입 `'a`에 대해서"라는 선언이 내포되어 있다고 생각하면 된다. 따라서 `combine`의 타입은 다음과 같이 해석할 수 있다.

- 아무 타입 `'a`에 대해서 `'a -> int` 타입의 함수 두개를 받아서 `'a -> int` 타입의 함수를 반환한다.

타입에 타입변수가 포함된 함수는 다형함수[polymorphic function]이라고 부른다.

연산식의 타입에는 여러 개의 타입변수가 이용될 수 있다. 예를 들어 다음 함수 `pair`의 타입에는 두개의 타입변수 `'a`와 `'b`가 이용되고 있다.

```
# let pair x y = (x, y);;
val pair : 'a -> 'b -> 'a * 'b = <fun>
```

여기서 인자 x와 y가 서로 다른 타입변수를 가지는 이유는 아무런 관계가 없는 두 인자는 같은 타입을 가질 필요가 없기 때문이다. 예를 들면 x가 int 타입을 가질 때 y는 똑같이 int 타입을 가져도 되지만 bool 타입을 가져도 되는 것이다.

```
# pair 0 0;;
- : int * int = (0, 0)
# pair 0 true;;
- : int * bool = (0, true)
```

pair의 타입은 다음과 같이 해석할 수 있다.

- 아무 타입 'a와 'b에 대해서 'a 타입의 인자와 'b 타입의 인자를 받아서 'a * 'b 타입의 튜플을 반환한다.

7 장

합집합

지금까지 이용한 자료형은 정수, 부동소수, 튜플, 함수 등과 같이 모두 OCAML언어에서 기본적으로 제공하는 자료형이다. 이번 강의에서는 새로운 자료형을 직접 정의하여 이용하는 방법을 배운다.

1 합집합과 패턴검사

컴퓨터의 두뇌에 해당하는 중앙연산장치(CPU)는 정수와 부동소수의 연산만을 수행할 수 있고 불린형 값이나 튜플과 같은 자료형은 직접 처리하지 않는다. 예를 들어 불린형 값 true와 false는 내부적으로 정수 1과 0으로 흉내냄으로써 구현하고 두개의 정수로 이루어진 튜플은 내부적으로 독립적인 정수 두개로 저장한다. 이렇게 컴퓨터를 이용한 모든 계산은 내부적으로는 정수와 부동소수의 연산으로 귀결된다.

그러나 컴퓨터가 최종적으로 수행하는 계산과 달리 사람이 표현하는 계산은 정수와 부동소수 외에 구체적인 의미를 가지는 자료형을 이용하면 이해하기 쉬울 수가 있다. 예로서 색깔, 요일, 정수의 대소 비교 결과를 생각해보자.

- 색깔을 계산에 이용하고자 할 때 Red, Green, Blue와 같은 단어를 직접 값으로 이용하면 이해하기 쉽다.
- 요일을 계산에 이용하고자 할 때 1부터 7까지의 정수를 요일로 해석하는 것보다 Sunday, Monday, $\cdots$, Saturday와 같은 단어를 직접 값으로 이용하면 이해하기가 더 쉽다.
- 두 정수의 대소 비교 결과는 세가지 가능성이 있다. 각각의 가능성을 -1, 0, 1 등의 정수로 흉내내는 것보다 Less, Equal, Greater와 같은 단어를 직접 이용한다면 이해하기가 더 쉽다.

사람이 이해할 수 있는 단어를 연산식으로 이용하면 전체 연산식을 이해하기 쉽다는 장점 외에 의미 없는 연산식을 미연에 방지할 수 있다는 또 다른 장점도 있다. 예를 들어 세가지 색깔 Red, Green, Blue를 정수 0, 1, 2로 흉내낸다고 가정하자. 이 경우 0, 1, 2는 상황에 따라서 실제 정수를 표현할 수도 있고 색깔 Red, Green, Blue를 표현할 수도 있으므로 연산식을 작성할 때 주의를 기울

여야 한다. 만약 실수로 Blue를 표현하는 정수 2에 1을 더했다면 이는 사실 의미없는 연산식이지만 OCAML은 이를 잘못된 연산식으로 해석하지 못한다. 그러나 Blue가 연산식으로 이용된다면 이는 Blue에 1을 더하는 연산식이 되므로 OCAML은 잘못된 연산식으로 판단한다.

이렇게 단어를 연산식으로 이용할 수 있게 하는 자료형이 합집합sum type이다. 합집합은 키워드 `type` 다음에 이름을 지정하고 생성자constructor라고 불리는 단어를 나열함으로써 정의한다.

`type` ⟨*합집합 이름*⟩ `=` ⟨*생성자*⟩$_1$ `|` ⟨*생성자*⟩$_2$ `|` $\cdots$ `|` ⟨*생성자*⟩$_n$ `;;`

각각의 생성자는 `|`로 분리되며 반드시 영어 대문자로 시작해야 한다. ⟨*생성자*⟩$_i$는 모두 상수연산식이며 타입은 ⟨*합집합 이름*⟩이 된다 ($1 \leq i \leq n$). 즉 타입 ⟨*합집합 이름*⟩은 ⟨*생성자*⟩$_1$, ⟨*생성자*⟩$_2$, $\cdots$, ⟨*생성자*⟩$_n$으로 이루어진 집합으로 해석할 수 있다.

다음은 합집합 `color`, `day`, `order`를 정의한 예이다.

- `type color = Red | Green | Blue;;`
- ```
  type day = Sunday | Monday | Tuesday | Wednesday | Thursday
           | Friday | Saturday;;
  ```
- `type order = Less | Equal | Greater;;`

다음은 위의 합집합 정의에 나열된 생성자를 이용한 연산식의 예이다.

```
# Red;;
- : color = Red
# (Red, Sunday);;
- : color * day = (Red, Sunday)
# (Red, Sunday, Less);;
- : color * day * order = (Red, Sunday, Less)
```

합집합 정의에 나열된 생성자는 상수연산식이므로 비교연산자 =을 이용하여 비교할 수 있다. 그러나 정수와 부동소수를 =을 이용하여 비교할 수 없는 것처럼 다른 합집합에 속한 생성자끼리는 비교를 할 수 없다.

```
# Red = Red;;
- : bool = true
# Red = Green;;
- : bool = false
# Red <> Green;;
- : bool = true
# Red = Sunday;;
This expression has type day but is here used with type color
```
```

합집합에 속하는 연산식의 결과를 검사하기 위해서는 새로운 형태의 패턴검사를 이용한다. (튜플을 위한 패턴검사와는 형태가 다르다.) n개의 생성자 $\langle$*생성자*$\rangle_i$를 가지는 합집합 T를 가정하자 ($1 \leq i \leq n$). 타입 T를 가지는 연산식 e의 결과를 검사하기 위해서는 키워드 `match`와 `with`로 시작되는 패턴검사연산식pattern matching expression을 이용한다.

`match` e `with`
| $\langle$*생성자*$\rangle_1$ `->` $\langle$*연산식*$\rangle_1$
| $\langle$*생성자*$\rangle_2$ `->` $\langle$*연산식*$\rangle_2$
$\cdots$
| $\langle$*생성자*$\rangle_n$ `->` $\langle$*연산식*$\rangle_n$

위의 패턴검사연산식은 e의 계산 결과가 패턴 $\langle$*생성자*$\rangle_i$와 일치하면 대응되는 연산식인 $\langle$*연산식*$\rangle_i$를 계산한다. e의 계산 결과는 패턴 $\langle$*생성자*$\rangle_1, \langle$*생성자*$\rangle_2, \cdots, \langle$*생성자*$\rangle_n$의 순서대로 비교된다.

다음은 `color` 타입의 변수 `x`를 분석하여 그 값이 `Red`이면 0, `Green`이면 1, `Blue`이면 2를 계산하는 패턴검사연산식의 예이다.

```
match x with
| Red -> 0
| Green -> 1
| Blue -> 2
```

패턴검사연산식은 분석되는 연산식의 계산 결과에 상관없이 최종 결과값이 결정될 수 있도록 모든 가능한 패턴을 포함해야 한다. 예를 들어 `color` 타입의 변수 `x`를 분석할 때 두 개의 패턴만 이용하는 경우를 생각해 보자.

```
match x with
| Red -> 0
| Green -> 1
```

만약 `x`의 값이 `Red`나 `Green`이라면 최종 결과값은 `0` 또는 `1`로서 정상적으로 결정된다. 그러나 `x`의 값이 우연히 `Blue`라면 일치하는 패턴이 없기 때문에 패턴검사는 실패하며 최종 결과값을 결정할 수가 없다. 따라서 예외상황이 발생한 뒤 계산은 비정상적으로 종료된다.

모든 가능한 패턴을 포함하지 않은 패턴검사연산식은 엄밀히 말하면 잘못된 연산식은 아니지만 분석되는 연산식에 따라서 일치하는 패턴이 없을 수가 있으므로 불완전한 연산식이다. OCAML은 이처럼 불완전한 패턴검사연산식이 발견되면 모든 가능한 패턴이 나열되지 않았음을 경고메시지로 출력해준다.

```
# let convert x =
    match x with
    | Red -> 0
    | Green -> 1;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Blue
val convert : color -> int = <fun>
```

위의 함수 convert는 불완전한 패턴검사연산식을 이용하여 color 타입의 값을 정수로 변환하고 있다. convert의 타입이 출력되기 전에 나오는 경고메시지는 밑줄 쳐진 연산식이 불완전한 패턴검사연산식이며 어떤 경우에 패턴검사가 실패하는지를 알려주고 있다.

변수를 패턴으로 이용하면 생성자의 개수보다 적은 수의 패턴을 이용하여 완전한 패턴검사연산식을 작성할 수 있다. 예를 들어 주어진 color 타입의 값이 Red인지 아닌지를 판별하는 함수를 생각해보자.

```
let is_red x =
  match x with
  | Red -> true
  | Green -> false
  | Blue -> false
```

is_red는 변수 x의 값을 Red, Green, Blue와 순서대로 비교한다. 그러나 Red가 아닌 경우에는 x의 실제값에 상관없이 false를 반환하면 된다. 이를 위해서 특수한 형태의 변수 _를 패턴으로 이용한다.

```
let is_red x =
  match x with
  | Red -> true
  | _ -> false
```

여기서 변수 _는 모든 값과 일치가 되는 패턴으로 이용되고 있다. 따라서 _ 다음에 추가로 패턴을 나열하는 것은 불필요하다. (이런 경우 OCAML은 불필요한 패턴이 있음을 경고메시지로 알려준다.)

만약 모든 값과 일치하는 패턴이 필요하고 그 값도 ->의 오른쪽 연산식에 이용하고자 하면 일반적인 변수를 패턴으로 이용하면 된다. 다음 함수는 주어진 color 타입의 값이 Red이면 Green으로 변환하고 다른 값은 그대로 반환한다.

```
let convert_red x =
  match x with
  | Red -> Green
  | y -> y
```

두번째 패턴검사에서는 변수 x의 값이 y라고 가정한 뒤 y를 결과값으로 취한다.

convert_red는 특수한 경우로서 x의 값을 굳이 y라고 다시 가정할 필요가 없이 x를 ->의 오른쪽에 직접 사용하면 된다.

```
let convert_red x =
  match x with
  | Red -> Green
  | _ -> x
```

그러나 이후 공부할 일반적인 합집합의 경우 _가 아닌 변수를 패턴으로 유용하게 이용할 수 있다.

2 인자를 가지는 합집합

지금까지 살펴본 합집합은 단순히 생성자들의 집합만을 나타내므로 그 표현력이 제한적이며 큰 집합을 나타내기에는 부적합하다. 예로서 합집합 color를 확장하여 색깔 종류 Red, Green, Blue뿐 아니라 색깔 강도도 함께 표현하는 새로운 합집합 icolor를 작성해보자. 색깔 강도는 0부터 255사이의 정수라고 가정하면 icolor는 총 256×3개의 생성자를 나열함으로써 정의할 수 있다.

```
type icolor = Red0 | Green0 | Blue0 | Red1 | Green1 | Blue1 |
              ··· | Red255 | Green255 | Blue255;;
```

이렇게 정의한 icolor는 너무 많은 생성자 때문에 사용하기 불편하다. 예를 들어 주어진 색깔이 Red인지 판별하기 위해서는 적어도 256개의 패턴을 나열하는 패턴검사연산식이 필요하다.

합집합의 표현력을 확장하기 위해서 OCAML은 생성자가 인자를 가지도록 허용한다. 생성자에 인자를 허용하기 위해서는 합집합을 정의할 때 키워드 of를 이용하여 인자의 타입을 지정해 주면 된다.

⟨*생성자*⟩ of ⟨*인자의 타입*⟩

생성자에 인자를 허용하면 다음과 같이 간단하게 icolor를 정의할 수 있다.

```
type icolor = Red of int | Green of int | Blue of int
```

이 정의에 따라서 생성자 Red, Green, Blue는 int 타입의 인자를 가진다. 따라서 이 생성자를 이용할 때 반드시 하나의 정수를 제공해야만 icolor 타입의 값을 만들 수 있다.

```
# Red 0;;
- : icolor = Red 0
# Green 10;;
- : icolor = Green 10
# Blue 0;;
- : icolor = Blue 0
```

합집합 icolor의 생성자는 모두 같은 타입의 인자를 가지지만 일반적으로 생성자들은 서로 다른 타입의 인자를 가지거나 인자를 아예 가지지 않아도 된다. 다음 합집합 acolor의 정의가 이를 보여준다.

```
type acolor = Rgb of int * int * int | Gray of int | Black
```

acolor의 정의에 따라 Rgb는 int * int * int 타입의 인자를 가지며 Gray는 int 타입의 인자를 가지며 Black은 인자를 가지지 않는다. Rgb와 Gray를 이용할 때는 정의에 명시된 타입의 연산식을 제공해야 한다.

```
# Rgb (100, 100, 100);;
- : acolor = Rgb (100, 100, 100)
# Gray 100;;
- : acolor = Gray 100
# Black;;
- : acolor = Black
```

인자가 있는 생성자를 패턴에 이용할 때는 생성자의 인자 위치에 또 다른 패턴을 적어주어야 한다. 다음 함수는 acolor 타입의 값을 int * int * int 타입의 튜플로 변환한다.

```
let rgb_of_acolor x =
  match x with
  | Rgb (r, g, b) -> (r, g, b)
  | Gray g -> (g, g, g)
  | Black -> (0, 0, 0)
```

첫번째 패턴검사는 변수 x의 값이 Rgb (r, g, b) 형태이면 정수 r을 변수 r에 저장하고, 정수 g를 변수 g에 저장하고, 정수 b를 변수 b에 저장한 뒤 ->의 오른쪽에 있는 연산식 (r, g, b)를 계산함을 나타낸다. 예로서 x의 값이 Rgb (0, 127, 255)이면 전체 패턴검사연산식의 계산 결과는 (0, 127, 255)가 된다. 두번째 패턴검사는 x의 값이 Gray g 형태이면 정수 g를 변수 g에 저장한 뒤 연산식 (g, g, g)를 계산함을 나타낸다. 세번째 패턴검사는 x의 값이 Black이면 (0, 0, 0)을 계산함을 나타낸다.

함수 rgb_of_acolor에서 첫번째 패턴과 두번째 패턴에 있는 같은 이름의 변수 g는 별도로 정의되는 패턴에 속하므로 서로 관계가 없다. 그러나 한 패턴 내에서는 같은 이름의 변수를 여러번 사용할 수 없다. 예로서 변수 x의 값이 Rgb (r, r, r) 형태인지 검사하기 위해서 패턴 Rgb (r, r, r)을 이용하면 오류가 발생한다. 이때는 다음 예처럼 Rgb (r, g, b) 패턴을 이용한 뒤 ->의 오른쪽에서 다시 변수 r, g, b가 같은 값을 가지는지 검사해야 한다.

```
let is_gray x =
  match x with
  | Rgb (r, g, b) -> r = g && g = b
  | _ -> true
```

3 다형합집합

합집합의 표현력을 확장하는 다른 방법을 알아보기 위해서 원소의 개수가 2 이하인 정수 집합을 표현하는 합집합 `int_sset`을 생각해보자 (`sset`은 "simple set"을 나타낸다).

```
type int_sset = Empty | Singleton of int | Pair of int * int;;
```

`Empty`는 공집합, `Singleton` n은 n이 유일한 원소인 집합, `Pair` (m, n)은 m과 n을 원소로 가지는 집합을 나타낸다. 그런데 원소의 개수가 2 이하인 부동소수 집합을 표현하는 합집합 `float_sset`도 동일한 형태로 정의할 수 있다.

```
type float_sset = Empty | Singleton of float | Pair of float * float;;
```

이 과정을 일반화하면 원소의 개수가 2 이하이고 같은 타입의 값으로만 이루어진 집합을 표현하기 위해서는 인자의 타입만 다르고 구조는 동일한 합집합을 매번 정의해야 함을 알 수 있다.

다형합집합polymorphic sum type은 생성자의 인자 타입을 구체적으로 명시하지 않고 타입변수로 지정하게 함으로써 인자의 타입만 다르고 구조는 동일한 모든 합집합을 하나의 정의로 완성하게 해 준다. 예를 들면 원소의 개수가 2 이하이고 같은 타입의 값으로만 이루어진 집합은 다음의 다형합집합으로 한꺼번에 표현할 수 있다.

```
type 'a sset = Empty | Singleton of 'a | Pair of 'a * 'a;;
```

키워드 `type` 뒤에 위치한 타입변수 `'a`는 생성자의 인자타입에 `'a`가 이용됨을 미리 알려준다. 3장의 다형함수와 마찬가지로 타입변수 `'a`가 포함된 다형합집합의 정의는 "아무 타입 `'a`에 대해서"라는 선언이 내포되어 있다고 생각하면 된다. 그러면 위의 정의는 다음과 같이 해석할 수 있다.

- 아무 타입 `'a`에 대해서 `Empty`는 `'a sset` 타입을 가진다.
- 아무 타입 `'a`에 대해서 연산식 e가 타입 `'a`를 가지면 `Singleton` e의 타입은 `'a sset`이다.
- 아무 타입 `'a`에 대해서 연산식 e가 타입 `'a * 'a`를 가지면 `Pair` e의 타입은 `'a sset`이다.

결국 위의 정의는 임의의 타입 T에 대해서 T `sset`이라는 합집합을 자동적으로 정의하는 효과를 가진다 (T와 `sset`는 붙어 있지 않고 띄어져 있다). 다음의 예는 `'a sset`의 생성자 세개를 모든 타입에 대해서 이용할 수 있음을 보여준다.

```
# Empty;;
- : 'a sset = Empty
# Singleton 1;;
- : int sset = Singleton 1
# Singleton 1.0;;
- : float sset = Singleton 1.
# Pair (1, 2);;
- : int sset = Pair (1, 2)
# Pair (1.0, 2.0);;
- : float sset = Pair (1., 2.)
# Pair (0, 1.0);;
This expression has type float but is here used with type int
```

첫번째 예는 `Empty`는 아무 타입 `'a`에 대해서 `'a sset` 타입을 가짐을 보여준다. 두번째와 세번째 예는 `Singleton`의 실제 인자에 따라서 연산식의 타입이 결정됨을 보여준다. 그 다음 두 예는 `Pair`의 실제 인자에 따라서 연산식의 타입이 결정됨을 보여준다. 마지막 예는 인자의 타입이 T `*` T 형태가 아니어서 `Pair`를 잘못 사용한 경우이다.

4 자기이용합집합

지금까지 살펴본 합집합은 비록 큰 집합을 간단하게 표현할 수는 있지만 크기가 무한인 집합은 표현하지 못한다. 예를 들어 `acolor`는 `int` 타입의 정수 개수가 유한하기 때문에 결국 유한집합을 나타낸다. 다형합집합 `'a sset`은 무한개의 합집합을 동시에 정의하는 효과를 가지지만 개별적인 합집합 T `sset`은 (타입 T가 유한집합을 나타내면) 역시 유한집합을 나타낸다.

합집합의 표현력을 확장하는 마지막 방법은 정의하는 합집합 자체를 생성자의 인자로 이용하여 크기가 무한인 집합을 표현하게 하는 것이다. 예로서 자연수의 집합을 합집합 `nat`으로 표현해보자. 자연수는 다음과 같은 성질을 가진다.

- 임의의 자연수는 0이거나 다른 자연수에 1을 더한 수이다.

자연수 0을 `Zero`로 나타내고 자연수 n에 1을 더한 자연수를 `Succ` n으로 나타낸다면 `nat`은 다음과 같이 정의된다 (`Succ`은 "successor"를 나타낸다).

```
type nat = Zero | Succ of nat;;
```

즉 합집합 `nat`의 생성자 `Succ`의 인자 타입으로 현재 정의되고 있는 합집합 `nat` 자체를 이용하는 것이다. 이렇게 정의된 합집합은 자기이용합집합[recursive sum type]이라고 부른다.

자기이용합집합의 생성자를 이용하는 방식은 기존의 합집합과 차이가 없다. 다음은 `nat`의 생성자를 이용한 연산식의 예이다.

```
# Zero;;
- : nat = Zero
# Succ Zero;;
- : nat = Succ Zero
# Succ (Succ Zero);;
- : nat = Succ (Succ Zero)
```

Zero는 자연수 0을, Succ Zero는 자연수 1을, Succ (Succ Zero)는 자연수 1에서 1을 더한 수인 자연수 2를 나타낸다. 이를 일반화하면 임의의 자연수 n은 Zero를 한번 이용하고 Succ을 n번 이용하여 나타낼 수 있다. 마지막 예에서 괄호를 생략하면 함수적용과 마찬가지로 (Succ Succ) Zero로 해석되어 오류가 발생된다.

자기이용합집합을 위한 패턴검사도 기존의 합집합과 차이가 없다. 다음 함수는 패턴검사를 이용하여 nat 타입의 인자가 Zero인지를 판별한다.

```
let is_zero n =
  match n with
  | Zero -> true
  | Succ _ -> false
```

여기서 변수 n의 값이 Succ p인 경우 p의 실제값은 중요하지 않기 때문에 Succ _을 패턴으로 이용하고 있다. 다음 함수는 nat 타입으로 주어진 두개의 자연수를 더한다.

```
let rec add m n =
  match m with
  | Zero -> n
  | Succ m' -> Succ (add m' n)
```

다음 함수는 비교연산자 =를 이용하지 않고 패턴검사만을 이용하여 nat 타입으로 주어진 두개의 자연수가 같은지 비교한다.

```
let rec equal m n =
  match m with
  | Zero -> (
    match n with
    | Zero -> true
    | _ -> false
    )
  | Succ m' -> (
    match n with
    | Zero -> false
    | Succ n' -> equal m' n'
    )
```

여기서 각각의 패턴이 속하는 패턴검사연산식을 명확히 결정하기 위해서 괄호를 이용해야 한다.

지금까지 패턴검사연산식은 합집합에 속한 값을 분석하기 위해서 생성자와 변수로 구성된 패턴만을 이용하였다. 그러나 2장에서 다룬 튜플을 분석하는 패턴과 자유롭게 조합하여 패턴검사연산식에 이용할 수 있다. 예를 들어 `nat * nat` 타입의 튜플을 분석하기 위해서 `(Zero, Zero)`, `(Zero, Succ n)`, `(Succ m, Zero)`, `(Succ m, Succ n)` 등의 패턴을 이용할 수 있다. 이런 형태의 튜플을 이용하여 `equal` 함수를 간단히 정의할 수 있다.

```
let rec equal m n =
  match (m, n) with
  | (Zero, Zero) -> true
  | (Succ m', Succ n') -> equal m' n'
  | (_, _) -> false
```

마지막 패턴은 `nat * nat` 타입의 모든 튜플과 일치하는 패턴이므로 간단히 _로 적어도 된다.

5 일반적인 형태의 합집합

합집합의 일반적인 형태는 타입변수를 이용하는 자기이용합집합이다. 예로서 원소들의 순서를 고려하는 집합인 리스트[list]를 다음과 같은 합집합으로 표현할 수 있다.

```
type 'a mylist = Nil | Cons of 'a * 'a mylist;;
```

생성자 `Nil`은 원소가 하나도 없는 빈 리스트를 나타낸다. 생성자 `Cons`는 튜플 (x, l)이 인자로 주어지면 x를 l의 맨 앞에 추가하여 생성된 리스트를 나타낸다. 예를 들면 다음 연산식은 정수 1, 2, 3으로 이루어진 리스트를 나타내며 `int mylist` 타입을 가진다.

```
Cons (1, Cons (2, Cons (3, Nil)))
```

리스트는 자주 이용되는 자료형이기 때문에 OCAML에서는 별도의 연산자와 함께 기본적으로 제공된다. OCAML에서 제공되는 리스트 자료형은 `'a list` 타입을 가지며 위의 `Nil`과 `Cons` 생성자를 이용하는 `'a mylist` 타입과는 무관하다.

8 장

리스트

리스트는 OCAML이 제공하는 기본적인 자료형으로서 같은 타입의 원소들이 순서대로 나열된 집합을 표현한다. 리스트는 타입변수를 이용하는 자기이용합집합으로 내부적으로 구현되어 있지만 7장에서 공부한 합집합 이용하는 방식보다 훨씬 간단하게 이용할 수 있다. 이번 강의에서는 OCAML이 제공하는 리스트를 이용하는 방법을 공부한다.

1 리스트의 생성

리스트를 생성하는 가장 간단한 방법은 리스트에 속한 원소들을 `[`과 `]`의 사이에 순서대로 나열하는 것이다. 각 원소들은 `;`로 분리하며 맨 마지막 원소 다음에 나오는 `;`는 무시된다. 원소가 하나도 없는 빈 리스트는 `[]`로 표시된다. 원소의 타입이 T이면 생성된 리스트는 T `list` 타입을 가진다.

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

리스트를 생성하는 다른 방법은 특정 원소를 이미 생성된 리스트의 맨 앞에 추가하는 것이다. 이를 위해서 이진연산자 `::`를 이용한다. h가 T 타입의 값이고 t가 T `list` 타입의 리스트일 때 h `::` t는 h를 t의 맨 앞에 추가한 리스트를 나타내며 T `list` 타입을 가진다 (h는 "head"를 나타내며 t는 "tail"을 나타낸다). `::`는 오른쪽으로 결합하므로 h_1 `::` h_2 `::` t와 h_1 `::` (h_2 `::` t)는 동등하다. `[`x_1`;` x_2`;` $\cdots$`;` x_n`]`과 x_1 `::` x_2 `::` $\cdots$ `::` x_n `::` `[]`는 같은 리스트를 나타낸다.

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

리스트를 생성하는 또 다른 방법은 두 리스트를 연결하는 것이다. 이를 위해서 이진연산자 @를 이용한다. l_1과 l_2가 같은 타입의 리스트일 때 l_1 @ l_2는 l_1과 l_2를 연결한 리스트를 나타낸다. @는 오른쪽으로 결합하므로 l_1 @ l_2 @ l_3와 l_1 @ (l_2 @ l_3)는 동등하다.

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

2 리스트의 패턴검사

임의의 리스트는 빈 리스트이거나 적어도 하나의 원소를 가지는 리스트이다. 패턴검사를 이용하여 리스트를 분석하고자 할 때는 이 두가지 가능성을 고려해야 한다.

- 빈 리스트인지 판별하기 위해서 패턴 `[]`을 이용한다.
- 적어도 하나의 원소를 가지는 리스트인지 판별하기 위해서 `h :: t` 형태의 패턴을 이용한다. 이 경우 분석되는 리스트가 적어도 하나의 원소를 가지면 첫번째 원소는 변수 `h`에 저장되고 두번째 이후의 원소로 이루어지는 리스트는 변수 `t`에 저장된다. 예를 들어 리스트 `[1; 2; 3]`은 패턴 `h :: t`와 일치하며 이 경우 변수 `h`와 `t`는 각각 `1`과 `[2; 3]`을 값으로 가진다.

다음 함수는 패턴검사를 이용하여 리스트의 길이를 계산한다.

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

실행 예는 다음과 같다.

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

다음 함수는 정수로 이루어진 리스트가 주어졌을 때 모든 원소의 합을 계산한다.

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

실행 예는 다음과 같다.

```
# sum [1; 2; 3];;
- : int = 6
```

다음 함수는 주어진 리스트의 원소 순서를 거꾸로 바꾼 리스트를 생성한다.

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

실행 예는 다음과 같다.

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

리스트를 분석할 때 `[]`과 `h :: t` 형태의 패턴을 기존의 패턴과 자유롭게 조합하여 이용할 수 있다. T `list` 타입의 리스트를 분석한다고 하자. `h :: t`는 이미 올바른 패턴이지만 `h`를 T 타입을 위한 임의의 패턴으로 대체해도 되며 `t`를 T `list` 타입을 위한 임의의 패턴으로 대체해도 된다. 예를 들면 `_ :: t`는 첫번째 원소는 상관하지 않는 패턴이며 `h1 :: h2 :: t`는 적어도 두개의 원소를 가지는 리스트와 일치하는 패턴이다 (패턴에서도 `::`는 오른쪽으로 결합하므로 `h1 :: h2 :: t`와 `h1 :: (h2 :: t)`는 동등하다).

다음 함수는 정수 리스트가 주어졌을 때 연속된 두 원소끼리의 합으로 이루어진 리스트를 생성한다. 이를 위해 여러 형태의 패턴을 이용한다.

```
let rec halve l =
  match l with
  | [] -> []
  | h1 :: [] -> [h1]
  | h1 :: h2 :: t -> (h1 + h2) :: halve t
```

실행 예는 다음과 같다.

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

9 장

계산과정의 설계

계산은 입력값으로부터 원하는 성질을 만족하는 결과값을 찾아내는 과정이다. 이러한 계산과정은 그 설계방식에 따라 계산에 걸리는 시간과 결과값의 정확도 등에 차이를 보일 수 있다. 이번 강의에서는 계산과정의 설계에 이용하는 방식 중 가장 많이 이용하는 두가지를 공부한다.

1 눈앞찾기

계산과정의 설계에 이용할 수 있는 가장 단순한 방식은 문제를 전체적으로 분석하지 않고 계산의 각 단계에 취할 수 있는 가장 쉬운 선택만을 쫓아가는 방식이다. 예를 들어 포항에서 서울까지의 최단 경로를 찾는다고 하자. 이 문제에 대한 최적의 답은 포항과 서울 사이의 모든 경로를 일일이 분석해야 찾을 수 있다. 만약 이 과정이 불가능하다면 서울 방향과 가장 가까운 길만을 따라가는 것이 현실적으로 좋은 대안이 된다. 이렇게 계산의 각 단계에서 가장 쉬운 선택만을 쫓아가는 방식을 눈앞찾기방식greedy approach 이라고 부른다.

눈앞찾기방식의 예로서 동전 개수를 최소화하면서 원하는 금액을 만드는 문제를 생각해 보자. 10원, 50원, 100원짜리 동전으로 160원을 만들려고 한다면 우선 100원짜리 동전 한개를 특별한 생각없이 선택할 것이다. 그 다음 나머지 60원을 50원짜리 동전 한개와 10원짜리 동전 한개로 만들 것이다. 이런 방식으로 동전을 선택하는 이유는 높은 금액의 동전이 전체 동전의 개수를 줄일 가능성이 높아 보이기 때문이다. 그리고 동전 두개의 조합으로는 160원을 만들 수가 없기 때문에 이렇게 찾은 조합이 사실 동전의 개수가 최소인 조합이다. 이렇게 가능한 높은 금액의 동전을 선택하는 것이 최적의 선택이라는 가정하에 눈앞찾기방식을 적용하여 문제를 해결할 수 있다.

눈앞찾기방식은 계산과정의 설계를 수월하게 해 주지만 문제에 따라서는 최적의 결과값을 보장하지 않을 수도 있다. 예를 들어 10원, 50원, 120원짜리 동전으로 160원을 만들려고 할 때 눈앞찾기방식을 적용하면 120원짜리 동전 한개와 10원짜리 동전 네개를 선택하게 된다. 그러나 50원짜리 동전 세개와 10원짜리 동전 한개를 선택하면 총 네개의 동전으로 160원을 만들 수 있다. 따라서 눈앞찾기방식으로 계산과정을 설계하고자 하면 계산의 결과값이 최적의 답인지 아닌지 별도로 분석해야 한다.

2 분할점령

분할점령방식divide and conquer은 주어진 문제를 크기가 작은 문제들로 나누고 각 문제의 답으로부터 원래 문제의 답을 유도하는 방식이다. 이 과정에서 크기가 작은 문제의 해결에 동일한 방식을 반복해서 적용하며, 문제의 크기를 더 이상 줄일 수 없을 때는 별도의 방식으로 직접 답을 찾아낸다. 이런 점에서 분할점령방식은 자기호출함수와 그 원리가 동일하다.

분할점령방식의 예로서 하노이의 탑tower of Hanoi 문제를 생각해 보자. 세 개의 막대 A, B, C가 있고 막대 A에는 n개의 원반이 크기 순서대로 꽂혀있다. 문제의 목표는 다음 조건을 유지하면서 막대 A에 꽂혀있는 원반을 막대 C로 모두 옮기는 순서를 찾아내는 것이다.

- 어느 시점에서도 큰 원반이 작은 원반의 위에 놓일 수 없다.

몇 가지 경우를 시험해 보면 다음 방법으로 원반을 모두 옮길 수 있음을 알 수 있다.

- 홀수번째는 제일 작은 원반을 시계방향으로 옮긴다.
- 짝수번째는 제일 작은 원반 외에 옮길 수 있는 원반이 유일하게 결정된다.

이 방법은 올바른 답을 제시하지만 왜 그렇게 되는지를 이해하기는 쉽지 않다. 그러나 분할점령방식을 적용하면 간단하고 이해하기 쉬운 답을 다음과 같이 구할 수 있다.

- n개의 원반중 위에 놓여 있는 $n-1$개의 원반을 다른 막대로 옮긴다.
- 맨 아래 놓여 있던 제일 큰 원반을 옮긴다.
- $n-1$개의 원반을 제일 큰 원반이 놓인 막대로 다시 옮긴다.

즉 n개의 원반을 옮기는 문제를 $n-1$개의 원반을 두번 옮기는 문제로 나누어 해결하고 있다. 이 때 $n-1$개의 원반을 옮기는 문제는 다시 $n-2$개의 원반을 두번 옮기는 문제로 나누어지며 이 과정은 한 개의 원반만 남을 때까지 반복된다. (원반이 한 개 남으면 그냥 다른 막대로 옮기면 된다.) 이렇게 분할점령방식으로 얻은 답을 분석해 보면 처음 소개한 답과 동일한 순서로 원반을 옮기는 것을 알 수 있다.

분할점령방식은 중요한 계산과정 설계방식으로서 여러 분야에서 광범위하게 이용하고 있다. 또한 분할점령방식에 기초한 계산과정을 효율적으로 구현하는 방법도 동적프로그래밍dynamic programming이란 이름으로 널리 알려져 있다. 지금까지 공부한 자기호출함수도 사실 그 원리가 분할점령방식과 동일하며 꼬리물기함수는 분할점령방식을 효율적으로 구현하는 방법으로 생각할 수 있다.

10 장

정렬

정렬[sorting]은 주어진 원소들을 크기 순서대로 나열하는 계산으로 가장 많이 이용하는 계산 중의 하나이다. 이번 강의에서는 정렬이라는 계산 문제를 눈앞찾기방식과 분할점령방식을 이용해서 여러가지 방법으로 해결해 본다.

1 눈앞찾기방식

크기를 비교할 수 있는 원소들의 리스트 $x_1, x_2, \cdots, x_n$을 정렬한다고 하자. 정렬의 결과는 다음 두 가지 조건을 만족하는 새로운 리스트 $y_1, y_2, \cdots, y_n$이다.

- $\{x_1, x_2, \cdots, x_n\} = \{y_1, y_2, \cdots, y_n\}$.
  즉 정렬은 처음 주어진 원소들의 위치만을 바꾼다.
- $y_1 \leq y_2 \leq \cdots \leq y_n$.
  즉 정렬의 결과로 나오는 리스트의 원소들은 순서대로 나열되어 있다.

눈앞찾기방식으로 정렬하고자 하면 계산의 각 단계에 취할 수 있는 가장 쉬운 선택이 무엇인지를 지정해 주면 된다. 선택정렬[selection sort]은 아직 정렬되지 않고 남아있는 원소들 중에서 가장 작은 원소를 *선택한* 뒤 정렬된 리스트의 맨 끝에 옮기는 방식이다. 이와 비슷한 삽입정렬[insertion sort]은 아직 정렬되지 않고 남아 있는 원소들 중 임의의 원소를 정렬된 리스트 내의 올바른 위치에 *삽입해* 가는 방식이다. 선택정렬과 삽입정렬은 정렬의 각 단계에서 이미 정렬된 리스트의 크기를 1 증가시킨다는 점에서 눈앞찾기 방식으로 간주할 수 있다.

1.1 선택정렬

선택정렬로 리스트 $x_1, x_2, \cdots, x_n$을 정렬한다고 하자. 계산의 초기 상태는 다음과 같다.

- 정렬되지 않고 남은 리스트는 $x_1, x_2, \cdots, x_n$이다.
- 이미 정렬된 리스트는 비어있다.

선택정렬은 남은 리스트에서 제일 작은 원소를 정렬된 리스트의 맨 뒤로 옮긴다. 이 과정을 남은 리스트가 빌 때까지 반복하면 처음 주어진 리스트를 정렬한 결과를 얻게 된다.

예를 들어 정렬되지 않고 남은 리스트 중에서 가장 작은 원소가 x_k라고 가정하자. (즉 $1 \leq i \leq n$에 대해서 $x_k \leq x_i$가 성립한다.) 선택정렬의 첫번째 단계는 x_k를 정렬된 리스트로 옮기고 남은 리스트에서 x_k를 삭제한다. 따라서 첫번째 단계 후 계산의 상태는 다음과 같다.

- 정렬되지 않고 남은 리스트는 $x_1, \cdots, x_{k-1}, x_{k+1}, \cdots, x_n$이다.
- 이미 정렬된 리스트는 x_k로 이루어져 있다.

일반적으로 선택정렬의 j번째 단계 후 계산의 상태는 다음과 같다 ($j \leq n$).

- 정렬되지 않고 남은 리스트는 $y_1, y_2, \cdots, y_{n-j}$이다.
- 이미 정렬된 리스트는 $y'_1, y'_2, \cdots, y'_j$이다.

이 때 남은 리스트와 정렬된 리스트는 다음 세가지 조건을 만족한다.

- $\{x_1, x_2, \cdots, x_n\} = \{y_1, y_2, \cdots, y_{n-j}, y'_1, y'_2, \cdots, y'_j\}$. 즉 남은 리스트와 정렬된 리스트는 처음에 주어진 원소들로만 이루어진다.
- $y'_1 \leq y'_2 \leq \cdots \leq y'_j$. 즉 $y'_1, y'_2, \cdots, y'_j$는 이미 정렬되어 있다.
- $1 \leq i \leq (n-j)$에 대해서 $y'_j \leq y_i$. 즉 y'_j는 남은 리스트에 있는 어떠한 원소보다 작거나 같다.

$(j+1)$번째 단계에서는 남은 리스트 $y_1, y_2, \cdots, y_{n-j}$에서 제일 작은 원소를 정렬된 리스트의 맨 뒤로 옮긴다. 이 과정은 n번째 단계까지 반복된다.

선택정렬을 OCAML로 구현한 예는 다음과 같다. `find_min` l은 리스트 l을 가장 작은 원소와 그 나머지 원소들의 리스트를 나눈다. `selection_sort` l은 선택정렬을 이용하여 리스트 l을 정렬한다.

```
let rec find_min l =
  match l with
  | h :: [] -> (h, [])
  | h :: t ->
    let (m, l') = find_min t in
    if h < m then (h, t)
    else (m, h :: l')

let rec selection_sort l =
  match l with
  | [] -> []
  | _ ->
    let (m, l') = find_min l in
    m :: (selection_sort l')
```

1.2 삽입정렬

삽입정렬로 리스트 $x_1, x_2, \cdots, x_n$을 정렬한다고 하자. 계산의 초기 상태는 선택정렬의 경우과 같다.

- 정렬되지 않고 남은 리스트는 $x_1, x_2, \cdots, x_n$이다.
- 이미 정렬된 리스트는 비어있다.

삽입정렬은 남은 리스트의 첫번째 원소를 정렬된 리스트의 올바른 위치에 삽입한다. 이 과정을 남은 리스트가 빌 때까지 반복하면 처음 주어진 리스트를 정렬한 결과를 얻게 된다. 이처럼 삽입정렬은 선택정렬과 비교했을 때 남은 리스트에서 원소를 하나 선별하는 과정이 더 단순하지만 정렬된 리스트에 원소를 추가하는 과정은 더 복잡하다.

삽입정렬의 첫번째 단계는 x_1을 정렬된 리스트로 옮기고 남은 리스트에서 x_1을 삭제한다. 따라서 첫번째 단계 후 계산의 상태는 다음과 같다.

- 정렬되지 않고 남은 리스트는 $x_2, \cdots, x_n$이다.
- 이미 정렬된 리스트는 x_1으로 이루어져 있다.

일반적으로 삽입정렬의 j번째 단계 후 계산의 상태는 다음과 같다 ($j \leq n$).

- 정렬되지 않고 남은 리스트는 $x_{j+1}, \cdots, x_n$이다.
- 이미 정렬된 리스트는 $y_1, y_2, \cdots, y_j$이다.

이 때 남은 리스트와 정렬된 리스트는 다음 두가지 조건을 만족한다.

- $\{x_1, x_2, \cdots, x_j\} = \{y_1, y_2, \cdots, y_j\}$. 즉 정렬된 리스트는 처음에 주어진 원소들로만 이루어진다.
- $y_1 \leq y_2 \leq \cdots \leq y_j$. 즉 $y_1, y_2, \cdots, y_j$는 이미 정렬되어 있다.

$(j+1)$번째 단계에서는 남은 리스트의 첫번째 원소 x_{j+1}을 정렬된 리스트의 올바른 위치에 삽입한다. 이 과정은 n번째 단계까지 반복된다.

삽입정렬을 OCAML로 구현한 예는 다음과 같다. `insert` l x는 이미 정렬된 리스트 l에 원소 x를 삽입하여 새로운 정렬된 리스트를 생성한다. `insertion_sort` l은 삽입정렬을 이용하여 리스트 l을 정렬한다.

```
let rec insert l x =
  match l with
  | [] -> [x]
  | h :: t ->
    if h < x then h :: insert t x
    else x :: h :: t
```

```
let insertion_sort l =
  let rec insertion_sort' sorted unsorted =
    match unsorted with
    | [] -> sorted
    | h :: t -> insertion_sort' (insert sorted h) t
    in
  insertion_sort' [] l
```

2 분할점령 방식

분할점령방식을 이용한 정렬은 원소들의 리스트를 두개의 작은 리스트로 나누고 각각의 리스트를 별도로 정렬한 뒤 다시 합친다. 분할점령방식의 대표적 예인 합병정렬merge sort은 두개의 리스트로 나누는 방법이 중요하지 않지만 정렬된 리스트를 합치는 과정이 특별하며, 분할점령방식의 또 다른 예인 퀵정렬quick sort은 두개의 리스트로 나누는 방법이 특별하지만 정렬된 리스트를 합치는 과정이 단순하다. 합병정렬과 퀵정렬은 선택정렬과 삽입정렬에 비해서 평균적으로 원소들의 비교회수가 적으며 실제로 가장 많이 이용되는 정렬 방법들이다.

2.1 합병정렬

합병정렬로 리스트 $x_1, x_2, \cdots, x_n$을 정렬한다고 하자. n이 0이거나 1이면 이미 정렬이 완료된 상태이다. n이 1보다 크다고 가정하자. 합병정렬의 첫번째 단계에서는 주어진 리스트를 다음 조건을 만족하는 두개의 작은 리스트 $y_1, y_2, \cdots, y_{n_1}$과 $z_1, z_2, \cdots, z_{n_2}$로 나눈다.

- $n_1 < n, n_2 < n, |n_1 - n_2| \leq 1$. 즉 크기가 줄어든 리스트가 생성되며 두 리스트의 길이는 같거나 1 차이가 난다.
- $\{x_1, x_2, \cdots, x_n\} = \{y_1, y_2, \cdots, y_{n_1}, z_1, z_2, \cdots, z_{n_2}\}$. 즉 생성된 두 리스트는 처음 주어진 원소들로만 이루어진다.

두 리스트는 위의 조건만 만족한다면 어떤 방식으로 생성해도 상관 없다. 두번째 단계에서는 생성된 두개의 리스트를 합병정렬을 이용하여 각각 정렬한다. 최종 단계에서는 정렬된 두 리스트의 원소들을 처음부터 순차적으로 비교해 가면서 크기 순서대로 원소들을 나열한다. 이 때 원소 비교를 한번 할 때마다 최종적으로 정렬된 리스트에 추가될 원소가 한개 결정되므로 총 n번의 원소 비교가 필요하다.

합병정렬을 OCAML로 구현한 예는 다음과 같다. `split` l은 리스트 l을 길이가 같거가 1 차이가 나는 두개의 리스트로 나누며 합병정렬의 첫번째 단계에서 이용한다. `merge` l_1 l_2는 이미 정렬된 리스트 l_1과 l_2를 합쳐서 새로운 정렬된 리스트를 만들며 합병정렬의 세번째 단계에서 이용한다. `merge_sort` l은 합병정렬을 이용하여 리스트 l을 정렬한다.

```
let rec split l =
  match l with
  | [] -> ([], [])
  | [h] -> ([h], [])
  | h :: t ->
    let (l1, l2) = split t in
    if List.length l1 <= List.length l2 then (h :: l1, l2)
    else (l1, h :: l2)

let rec merge l1 l2 =
  match (l1, l2) with
  | ([], _) -> l2
  | (_, []) -> l1
  | (h1 :: t1, h2 :: t2) ->
    if h1 < h2 then h1 :: (merge t1 l2)
    else h2 :: (merge l1 t2)

let rec merge_sort l =
  match l with
  | [] -> []
  | [h] -> [h]
  | _ ->
    let (l1, l2) = split l in
    let l1' = merge_sort l1 in
    let l2' = merge_sort l2 in
    merge l1' l2'
```

2.2 퀵정렬

퀵정렬로 리스트 $x_1, x_2, \cdots, x_n$을 정렬한다고 하자. 문제를 간단하게 하기 위해서 모든 원소는 서로 다르다고 가정하자. n이 0이거나 1이면 이미 정렬이 완료된 상태이므로 n이 1보다 큰 경우만 고려하면 된다. 퀵정렬의 첫번째 단계에서는 기준원소라는 특별한 원소를 하나 지정한 뒤 기준원소보다 작은 원소들로 이루어진 리스트와 기준원소보다 큰 원소들로 이루어진 리스트로 나눈다. 예를 들어 기준원소를 x_1로 지정한다면 생성된 리스트 $y_1, y_2, \cdots, y_{n_1}$과 $z_1, z_2, \cdots, z_{n_2}$는 다음 조건을 만족해야 한다.

- $1 \leq i \leq n_1$에 대해서 $y_i < x_1$. 즉 첫번째 리스트는 기준원소 x_1보다 작은 원소들로 이루어진다.
- $1 \leq i \leq n_2$에 대해서 $x_1 < z_i$. 즉 두번째 리스트는 기준원소 x_1보다 큰 원소들로 이루어진다.

- $\{x_2, \cdots, x_n\} = \{y_1, y_2, \cdots, y_{n_1}, z_1, z_2, \cdots, z_{n_2}\}$. 즉 생성된 두 리스트는 처음 주어진 리스트 중에서 기준원소를 제외한 원소들로만 이루어진다.

두번째 단계에서는 생성된 두개의 리스트를 퀵정렬을 이용하여 각각 정렬한다. 이렇게 정렬된 두 리스트를 $y'_1, y'_2, \cdots, y'_{n_1}$과 $z'_1, z'_2, \cdots, z'_{n_2}$이라고 하자. 최종 단계에서는 정렬된 두 리스트와 기준원소를 합쳐서 새로운 리스트 $y'_1, y'_2, \cdots, y'_{n_1}, x_1, z'_1, z'_2, \cdots, z'_{n_2}$을 생성한다.

퀵정렬에서 기준원소를 정하는 방법은 정해져 있지 않으며 따라서 생성된 두 리스트의 길이에 대한 추가적인 조건은 없다. 극단적인 경우로 제일 작은 원소를 기준원소로 항상 정하는 경우 퀵정렬은 선택정렬과 차이가 없게 된다. 그러나 이런 특수한 경우는 드물며 평균적으로 퀵정렬은 제일 빠른 정렬방법 중의 하나로 알려져 있다.

퀵정렬을 OCAML로 구현한 예는 다음과 같다. `pivot_split` p l은 리스트 l을 기준원소 p보다 작은 원소로 이루어지는 리스트와 큰 원소로 이루어지는 리스트로 나눈다. `quick_sort` l은 퀵정렬을 이용하여 리스트 l을 정렬한다.

```
let pivot_split pivot l =
  (List.filter (fun y -> y < pivot) l,
   List.filter (fun y -> pivot <= y) l)

let rec quick_sort l =
  match l with
  | [] -> []
  | h :: t ->
    let (l1, l2) = pivot_split h t in
    (quick_sort l1) @ [h] @ (quick_sort l2)
```

11 장

나무

계산은 '무엇'을 '어떻게' 처리하여 결과값을 얻는지 설명하는 것으로 이루어진다. 따라서 '무엇'이 고정되어 있을 때는 '어떻게'에 따라서 계산의 속도와 효율성이 결정된다. 예를 들어서 정렬 문제의 경우 원소들의 리스트라는 고정된 '무엇'이 주어지고 선택정렬, 삽입정렬, 합병정렬, 퀵정렬 등과 같이 속도와 효율성이 다른 여러가지 '어떻게' 방식이 존재한다.

그러나 '무엇'이 고정되어 있지 않은 경우에는 '무엇'의 설계 방식에 따라서 계산의 속도와 효율성이 미리 결정되어 버릴 수도 있다. 예로서 집합 $S = \{x_1, x_2, \cdots, x_n\}$에 특정 원소 x가 포함되어 있는지 여부를 결정하는 계산을 생각해 보자. 만약 S를 단순히 리스트 $x_1, x_2, \cdots, x_n$으로 표현한다면 어떠한 '어떻게' 방식을 이용하더라도 최악의 경우 n번의 원소간 비교가 필요하다 (x가 S에 속하지 않는 경우). 그러나 최악의 경우에도 n번 미만의 원소간 비교가 일어나도록 S를 표현하는 방식도 존재한다.

이번 강의에서는 '무엇'의 설계 방식이 계산의 속도와 효율성에 영향을 미치는 것을 배우기 위해 집합을 표현하는 새로운 방식인 나무tree를 공부한다.

1 나무의 정의

나무는 리스트처럼 집합을 표현하는 자료형으로서 잎leaf과 가지node로 이루어진다. 잎은 그 자체로 나무이며 가지는 여러개의 작은 나무를 포함한다. 나무의 특수한 경우로서 가지가 항상 두개의 작은 나무를 포함하면 이진나무binary tree라고 불린다. 나무의 개념을 이해하는데는 이진나무로서 충분하므로 이후의 모든 설명은 이진나무만을 다루며 앞으로 나무는 이진나무를 지칭한다고 약속한다.

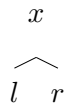
나무는 타입변수를 이용하는 자기이용합집합으로 정의할 수 있다. 이 때 결정해야 할 사항은 잎에 원소를 저장하는지 여부이다. 우선 잎에 원소를 저장하는 경우를 고려해 보자.

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

- `Leaf` x는 원소 x를 저장하는 잎을 나타낸다.

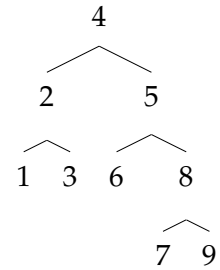
- `Node (`l`, `x`, `r`)`은 왼쪽에는 작은 나무 l, 오른쪽에는 작은 나무 r, 그리고 자신은 원소 x를 저장하는 가지를 나타낸다.

`Leaf` x $\Longleftrightarrow$ x

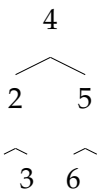
`Node (`l`, `x`, `r`)` $\Longleftrightarrow$ 

다음은 잎에 원소를 저장하는 나무의 예이다.

```
Node (
 Node (Leaf 1, 2, Leaf 3),
 4,
 Node (Leaf 6, 5,
   Node (Leaf 7, 8, Leaf 9)))
```

$\Longleftrightarrow$ 

잎에 원소를 저장하는 나무는 유용한 경우가 많지만 일반적인 형태의 나무를 표현하지는 못한다. 왜냐하면 가지는 항상 두개의 작은 나무를 포함하므로 왼쪽이나 오른쪽이 비어있는 나무를 표현하지 못하기 때문이다. 예를 들어 다음의 나무는 위의 합집합으로 표현할 수 없다.

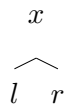


모든 형태의 나무를 표현하고자 하면 잎에 원소를 저장하지 않으면 된다.

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

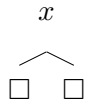
- `Leaf`는 (원소를 저장하지 않는) 잎을 나타낸다. 그림으로는 □로 나타낸다.
- `Node (`l`, `x`, `r`)`은 왼쪽에는 작은 나무 l, 오른쪽에는 작은 나무 r, 그리고 자신은 원소 x를 저장하는 가지를 나타낸다.

`Leaf` $\Longleftrightarrow$ □

`Node (`l`, `x`, `r`)` $\Longleftrightarrow$ 

`Leaf`는 원소를 저장하지 않으므로 원소 x만을 저장하는 나무는 다음과 같이 표현한다.

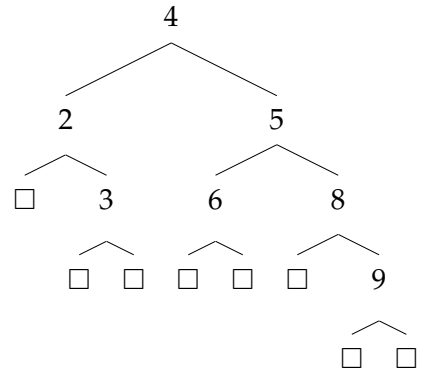
Node(Leaf, x, Leaf) $\Longleftrightarrow$



다음은 잎에 원소를 저장하지 않는 나무의 예이다.

```
Node (
 Node (Leaf, 2, Node (Leaf, 3, Leaf)),
 4,
 Node (Node (Leaf, 6, Leaf),
   5,
   Node (Leaf, 8, Node (Leaf, 9, Leaf))))
```

$\Longleftrightarrow$

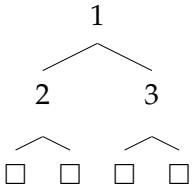


이후의 모든 설명은 잎에 원소를 저장하지 않는 나무를 가정한다.

2 나무의 원소 나열

나무는 정수, 부동소수, 리스트처럼 일종의 자료형이므로 여러가지 연산을 생각해 볼 수 있다. 나무에 적용할 수 있는 연산의 예로서 나무의 원소를 나열하여 리스트를 생성하는 연산을 생각해 보자.

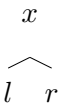
나무에 저장되어 있는 원소를 나열하는 방법은 유일하지 않다. (원소를 나열하는 방법이 유일하다면 나무와 리스트를 구별할 필요가 없을 것이다.) 예로서 세개의 원소를 저장하는 다음 나무를 생각해 보자.



뿌리에 저장되어 있는 원소 1을 나열하는 순서에 따라서 다음의 결과를 얻는다.

- 1을 제일 먼저 나열하면 1,2,3 또는 1,3,2의 리스트가 생성된다.
- 1을 중간에 나열하며 2,1,3 또는 3,1,2의 리스트가 생성된다.
- 1을 제일 나중에 나열하면 2,3,1 또는 3,2,1의 리스트가 생성된다.

이렇게 뿌리에 저장되어 있는 원소를 기준으로 삼고 왼쪽의 작은 나무를 오른쪽의 작은 나무보다 먼저 처리한다고 가정하면 다음과 같은 세가지 원소 나열 방식이 결정된다. 아래 설명에서는 뿌리에 x가 저장되어 있고 왼쪽 작은 나무가 l이며 오른쪽 작은 나무가 r인 나무를 이용한다:



- 앞순위preorder 나열 방식은 x를 먼저 나열한 뒤 l과 r의 원소를 앞순위로 나열한다.
- 안순위inorder 나열 방식은 l의 원소를 안순위로 나열한 뒤 x를 나열하고 r의 원소를 안순위로 나열한다.
- 뒷순위postorder 나열 방식은 l과 r의 원소를 뒷순위로 나열한 뒤 x를 맨 마지막에 나열한다.

나무의 원소 나열 방식을 OCAML로 구현한 예는 다음과 같다. `Leaf`는 원소를 포함하지 않으므로 빈 리스트를 반환한다.

앞순위 나열:

```
let rec preorder t =
  match t with
  | Leaf -> []
  | Node (l, v, r) -> v :: preorder l @ preorder r
```

안순위 나열:

```
let rec inorder t =
  match t with
  | Leaf -> []
  | Node (l, v, r) -> inorder l @ [v] @ inorder r
```

뒷순위 나열:

```
let rec postorder t =
  match t with
  | Leaf -> []
  | Node (l, v, r) -> postorder l @ postorder r @ [v]
```

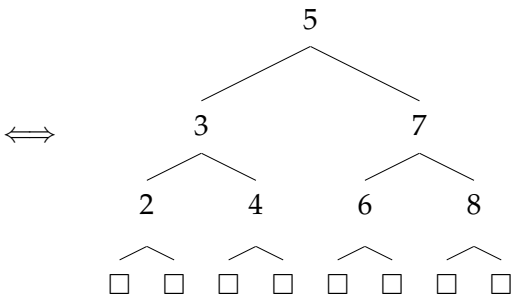
1절 73 페이지의 나무에 각각의 원소 나열 방식을 적용한 결과는 다음과 같다.

- 앞순위 나열: `[4; 2; 3; 5; 6; 8; 9]`
- 안순위 나열: `[2; 3; 4; 6; 5; 8; 9]`
- 뒷순위 나열: `[3; 2; 6; 9; 8; 5; 4]`

3 이진검색나무

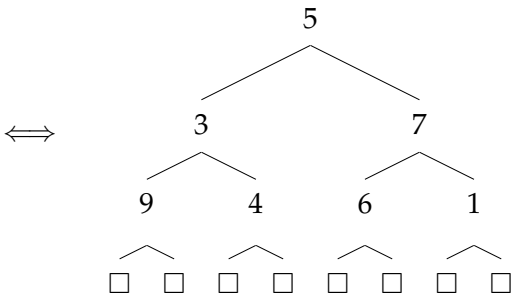
이진검색나무binary search tree는 이진나무의 특수한 형태로서 왼쪽 작은 나무의 모든 원소는 뿌리에 저장된 원소보다 작고 오른쪽 작은 나무의 모든 원소는 뿌리에 저장된 원소보다 큰 성질을 가진다. 다음은 이진검색나무의 예이다.

```
Node (
 Node (
   Node (Leaf, 2, Leaf),
   3,
   Node (Leaf, 4, Leaf)),
 5,
 Node (
   Node (Leaf, 6, Leaf),
   7,
   Node (Leaf, 8, Leaf)))
```



다음 나무는 원소 9와 원소 1이 이진검색나무의 요구 조건을 위반하므로 이진검색나무가 아니다.

```
Node (
 Node (
   Node (Leaf, 9, Leaf),
   3,
   Node (Leaf, 4, Leaf)),
 5,
 Node (
   Node (Leaf, 6, Leaf),
   7,
   Node (Leaf, 1, Leaf)))
```



이진검색나무의 요구 조건은 나무의 모든 부분에 적용된다. 따라서 `Node (`l`, `x`, `r`)`이 이진검색나무라면 l과 r도 이진검색나무이다.

이진검색나무는 그 이름이 의미하듯이 특정 원소를 검색할 때 유용하다. 예를 들어 이진검색나무 t에서 특정 원소 x를 찾는 연산을 생각해 보자. 이진검색나무의 특성을 고려하면 다음과 같이 t에서 x를 찾을 수 있다.

- t `= Leaf`이면 x는 t에 없다.
- t `= Node (`l`, `y`, `r`)`이고 $x = y$인 경우 x는 t의 뿌리에 저장되어 있다.
- t `= Node (`l`, `y`, `r`)`이고 $x < y$이면 x는 l에서 찾으면 된다.
- t `= Node (`l`, `y`, `r`)`이고 $x > y$이면 x는 r에서 찾으면 된다.

이진검색나무가 원소 검색에 유용한 이유는 불필요한 원소간 비교가 일어나지 않아서 전체 비교 회수가 작기 때문이다. 예를 들어 위의 이진검색나무는 나무에 저장된 원소를 검색할 때는 최대 3번의 원소간 비교가 필요하며 나무에 저장되어 있지 않은 원소를 검색할 때는 정확하게 3번의 원소간 비교가 필요하다. 일반적으로 n개의 원소를 저장하는 이진검색나무는 나무에 저장된 원소를 검색할 때는 최대 $\lceil \log_2 n \rceil$번, 나무에 저장되어 있지 않은 원소를 검색할 때는 정확하게 $\lceil \log_2 n \rceil$번의 원

소간 비교가 필요하도록 설계할 수 있다. 반면에 길이 n의 리스트에서 특정 원소를 검색할 때는 리스트에 포함된 원소의 경우는 최대 n번, 리스트에 포함되어 있지 않은 원소를 검색할 때는 정확하게 n번의 원소간 비교가 필요하다.

이진검색나무의 원소 검색을 위한 연산을 OCAML로 구현해 보자. `'a tree` 타입의 나무의 검색 결과를 나타내기 위해서 다음의 합집합을 이용한다.

```
type 'a option = Some of 'a | None;;
```

`Some` x는 원소 x를 찾았음을 의미하며 `None`은 찾는 원소가 나무에 저장되어 있지 않음을 의미한다. `'a option` 타입을 이용하면 `'a tree` 타입의 나무 검색은 `'a tree -> 'a -> 'a option` 타입의 함수 `search`로 구현할 수 있다. `search` t x는 이진검색나무 t에서 원소 x를 검색한 결과를 반환한다.

```
let rec search t x =
  match t with
  | Leaf -> None
  | Node (l, y, r) ->
    if x = y then Some x
    else if x < y then search l x
    else search r x
```

이진검색나무를 생성하기 위해서는 새로운 원소를 추가하는 연산을 구현하면 된다. 예를 들어 1, 2, 3, 4를 저장하는 이진검색나무를 생성하기 위해서는 `Leaf`에 1, 2, 3, 4를 아무 순서로 추가하면 된다. (원소를 추가하는 순서에 따라서 다른 이진검색나무가 생성될 수 있음에 유의하자.) 이진검색나무 t에 원소 x를 추가하는 연산은 다음과 같이 구현할 수 있다. 각각의 경우 연산의 결과로 생성되는 나무는 이진검색나무의 요구 조건을 만족한다.

- t = `Leaf`이면 `Node (Leaf,` x`, Leaf)`를 생성한다.
- t = `Node (`l`,` y`,` r`)`이고 $x = y$인 경우 t를 그대로 반환한다.
- t = `Node (`l`,` y`,` r`)`이고 $x < y$이면 x를 l에 추가한 결과를 l'이라고 할 때 `Node (`l'`,` y`,` r`)`을 반환한다.
- t = `Node (`l`,` y`,` r`)`이고 $x > y$이면 x를 r에 추가한 결과를 r'이라고 할 때 `Node (`l`,` y`,` r'`)`을 반환한다.

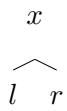
이진검색나무에 원소를 추가하는 함수를 OCAML로 구현한 예는 다음과 같다. `insert`는 `'a tree -> 'a -> 'a tree` 타입을 가지며 `insert` t x는 이진검색나무 t에 원소 x를 추가한 결과를 계산한다.

```
let rec insert t x =
  match t with
  | Leaf -> Node (Leaf, x, Leaf)
  | Node (l, y, r) ->
    if x = y then Node (l, y, r)
    else if x < y then Node (insert l x, y, r)
    else Node (l, y, insert r x)
```

이진검색나무의 또 다른 연산 예로 지정된 원소를 삭제하는 연산을 생각해 보자. 이진검색나무 t에서 원소 x를 삭제할 때 다음의 세가지 경우는 쉽게 처리할 수 있다.

- t = Leaf이면 t에 x가 없으므로 Leaf를 반환한다.
- t = Node (l, y, r)이고 $x < y$이면 l에서 x를 삭제한 결과를 l'이라고 할 때 Node (l', y, r)을 반환한다.
- t = Node (l, y, r)이고 $x > y$이면 r에서 x를 삭제한 결과를 r'이라고 할 때 Node (l, y, r')을 반환한다.

가장 복잡한 경우는 t = Node (l, y, r)이고 $x = y$인 경우이다. 즉 다음 나무에서 뿌리에 있는 원소 x를 삭제하는 경우이다.



이 경우 l이나 r의 형태에 따라서 처리 방식이 달라진다. 여기서는 l의 형태를 분석해서 x를 삭제해 보자.

우선 l = Leaf인 경우를 생각해 보자. x를 삭제한 뒤 남는 원소는 오른쪽 작은 나무 r에 저장된 원소이다. r은 그 자체로 이진검색나무이므로 x를 삭제한 결과로서 r을 취할 수 있다.

l = Leaf가 아닌 경우를 생각해 보자. x가 삭제되면 나무의 뿌리에 새로운 원소를 저장해야 하는데 이 원소는 l에서 가장 큰 원소를 취하면 된다. 즉 l에 있는 가장 큰 원소를 l_{max}라 하고 $l - l_{max}$는 l에서 l_{max}를 삭제한 나무를 나타낸다고 하면 t에서 x를 삭제한 결과로 Node ($l - l_{max}$, l_{max}, r)을 취할 수 있다. 여기서 l은 하나 이상의 원소를 저장하므로 l_{max}는 반드시 존재한다.

이진검색나무에서 원소를 삭제하는 함수를 OCAML로 구현한 예는 다음과 같다. delete_max는 'a tree -> 'a tree * 'a 타입을 가지며 Leaf가 아닌 이진검색나무를 인자로 받아들

인다. `delete_max` t는 t에 있는 가장 큰 원소와 이 원소를 삭제한 이진검색나무를 계산한다. `delete`는 `'a tree -> 'a -> 'a tree` 타입을 가지며 `delete` t x는 이진검색나무 t에서 x를 삭제한 결과를 계산한다.

```
let rec delete_max t =
  match t with
  | Node (l, x, Leaf) -> (l, x)
  | Node (l, x, r) ->
    let (r', m) = delete_max r in
    (Node (l, x, r'), m)

let rec delete t x =
  match t with
  | Leaf -> Leaf
  | Node (l, y, r) ->
    if x < y then Node (delete l x, y, r)
    else if x > y then Node (l, y, delete r x)
    else
      match (l, r) with
      | (Leaf, _) -> r
      | (Node _, _) ->
        let (l', m) = delete_max l in
        Node (l', m, r)
```

12 장

허프만 나무

지금까지 계산과정을 설계할 때 눈앞찾기방식과 분할점령방식을 이용할 수 있고 나무를 이용하여 집합을 표현할 수 있음을 배웠다. 이번 강의에서는 눈앞찾기방식과 나무를 응용 예로서 정보를 압축하는데 이용하는 허프만 나무Huffman tree를 공부한다.

1 접두 코드

컴퓨터에 저장되는 모든 정보는 최종적으로 0과 1로 이루어진 이진수로 변환된다. 이는 컴퓨터의 메모리가 0과 1의 연속으로 이루어져 있기 때문이다. 따라서 정보를 저장하기 위해서는 문자, 숫자, 특수문자와 같은 개별 정보를 이진수로 변환시키는 체계가 필요하며 이러한 변환 체계를 코드code라고 부른다. 그리고 주어진 코드를 이용하여 정보를 이진수로 변환하는 과정을 인코딩encoding, 이진수에서 정보를 복원하는 과정을 디코딩decoding라고 한다.

코드의 대표적인 예는 아스키코드ASCII code이다. 아스키코드는 256개의 문자, 숫자, 특수문자를 8자리의 이진수로 변환한다.[1] 예를 들어 영문자 A는 이진수 01000001로 변환하며 숫자 1은 이진수 00110001로 변환한다. 아스키코드는 모든 개별 정보를 8자리의 이진수로 변환하기 때문에 인코딩 과정에서는 생성된 각각의 이진수를 별도의 구분문자 없이 그냥 연결하기만 하면 된다. 예를 들어 A1A라는 정보는 01000001, 00110001, 01000001를 연결한 010000010011000101000001로 인코딩된다. 디코딩 과정은 8자리 단위로 처리하므로 이 24자리 이진수는 다시 A1A로 복원된다.

아스키코드의 경우 모든 개별 정보를 고정된 자리수의 이진수로 변환하므로 디코딩 과정에서 애매한 상황이 발생하지 않는다. 그러나 개별 정보를 길이가 다른 이진수로 변환하는 경우에는 별도의 구분문자가 필요할 수도 있다. 예로서 문자 A, B, C, D, E를 다음과 같이 변환하는 코드를 생각해 보자.

$$\mathrm{A} \to 1, \quad \mathrm{B} \to 00, \quad \mathrm{C} \to 01, \quad \mathrm{D} \to 000, \quad \mathrm{E} \to 0001$$

이 경우 0001은 BC, DA, E 중 어떤 것으로도 디코딩 될 수 있다. 따라서 인코딩 과정은 00/01, 000/1, 0001 등과 같이 별도의 구분문자를 이용하여 디코딩 과정에서 애매한 상황이 발생하지 않도

[1] 최초의 아스키코드는 128개의 문자, 숫자, 특수문자만을 변환한다.

록 해야한다.

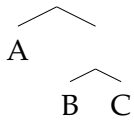
그러면 디코딩 과정에서 애매한 상황이 발생하는 않게 하려면 어떻게 코드를 설계해야 할까? 위의 코드를 살펴보면 B를 변환한 00은 D를 변환한 000의 앞부분과 중복되며, D를 변환한 000은 E를 변환한 0001의 앞부분과 중복됨을 알 수 있다. 따라서 00까지 주어졌다면 B로 디코딩 해야 할지, 아니면 다음에 나오는 숫자를 검사해야 하는지 바로 결정할 수가 없다. 만약 00 다음에 1이 나오면 00을 B로 해석해야 하지만 0이 나오면 그 다음에 나오는 숫자를 또 다시 검사해야 한다. 결과적으로 디코딩 과정에서 애매한 상황을 피하려면 변환된 이진수 중의 어느 것도 다른 이진수의 첫 부분과 중복되지 않도록 코드를 설계해야 한다.

변환된 이진수 중의 어느 것도 다른 이진수의 첫 부분과 중복되지 않는 코드를 접두코드prefix code라고 부른다. 다음은 문자 A, B, C, D, E를 변환하는 접두코드의 예이다.

$$A \to 000, \quad B \to 001, \quad C \to 01, \quad D \to 10, \quad E \to 11$$

접두코드를 이용하면 디코딩 과정에서 애매한 상황이 발생하지 않으므로 인코딩 과정에서 별도의 구분문자가 필요하지 않다. 예를 들어 ABCDE는 000001011011로 변환된다.

접두코드를 설계하는 방법으로 잎에만 원소가 저장된 이진나무를 이용하는 방법이 있다. (가지에는 원소가 저장되지 않음에 주의한다.) 예로서 다음의 이진나무를 생각해 보자.

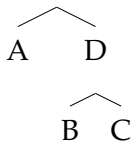


이 나무의 뿌리에서 각각의 잎까지 가는 경로는 하나의 이진수에 대응된다. 즉 왼쪽 작은 나무를 택할 때마다 0을, 오른쪽 작은 나무를 택할 때마다 1을 추가해 가면 잎에 도달할 때 유일한 이진수가 결정된다. 이 이진수는 뿌리에서 잎까지 가는 유일한 경로를 표현하며 잎에 저장된 원소를 변환하는 접두코드의 설계에 이용할 수 있다. 위 나무는 다음의 접두코드를 생성한다.

$$A \to 0, \quad B \to 10, \quad C \to 11$$

- 뿌리에서 A에 도달하기 위해서는 왼쪽 작은 나무를 택한다 (0).
- 뿌리에서 B에 도달하기 위해서는 오른쪽 작은 나무를 택한 뒤 (1) 왼쪽 작은 나무를 택한다 (0).
- 뿌리에서 C에 도달하기 위해서는 오른쪽 작은 나무를 택한 뒤 (1) 오른쪽 작은 나무를 택한다 (1).

이렇게 이진나무로부터 생성된 코드가 접두코드인 이유는 가지에 원소가 저장되지 않기 때문이다. 가지에도 원소가 저장된 다음 나무를 생각해 보자.



이 경우 B와 C에 도달하기 위해서는 반드시 D를 거쳐야 하므로 D를 변환한 이진수는 B와 C를 변환하는 이진수의 앞부분과 중복된다. 따라서 결과로서 생성된 코드는 접두코드가 아니다.

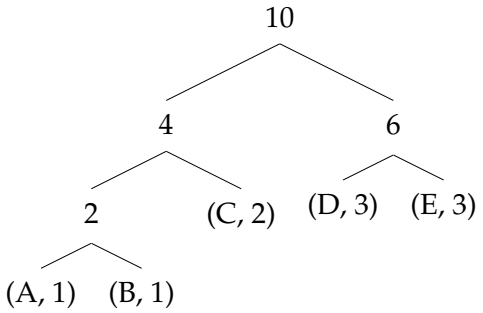
위에서 본 바와 같이 접두코드를 설계하는 문제는 가지에 원소가 저장되지 않고 잎에만 원소가 저장된 이진나무를 만드는 문제로 귀결된다. 다음 절에서 공부하는 허프만 나무는 원소의 빈도를 고려하여 정보를 압축하여 저장할 수 있게 하는 접두코드를 만들어 낸다.

2 허프만 나무

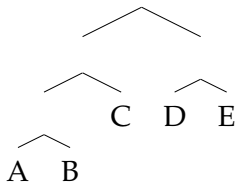
허프만 나무는 원소의 상대적 빈도가 고정된 정보를 압축해서 저장하고자 할 때 이용하는 이진나무이다. 예로서 문자 A, B, C, D, E로 이루어진 문자열을 이진수로 압축하고자 한다고 하자. 문자의 빈도는 다음과 같다.

$$A : 1, B : 1, C : 2, D : 3, E : 3$$

즉 A가 한번 나올 때 평균적으로 B도 한번만 나오며, C는 두번, D와 E는 세번 나온다. 이렇게 문자의 빈도가 주어졌을 때 다음과 같이 잎에는 각 문자와 빈도를, 가지에는 빈도의 합을 저장한 허프만 나무가 생성된다.



이 허프만 나무는 다음의 이진나무와 같은 구조를 가진다.



위의 허프만 나무에서 생성된 접두코드는 다음과 같다.

$$A \to 000, \quad B \to 001, \quad C \to 01, \quad D \to 10, \quad E \to 11$$

이렇게 허프만 나무에서 만들어진 접두코드를 허프만 코드Huffman code라고 부르며 허프만 코드를 생성하는 과정을 허프만 코딩Huffman coding이라고 부른다. 허프만 코드는 원소의 발생 빈도가 고정되어 있을 때 최대의 압축율을 보장하는 접두코드이다.

허프만 나무는 눈앞찾기 방식에 기초하여 만든다. 즉 빈도가 높은 원소를 가능한 짧은 이진수로 변환하여 정보의 압축률을 높인다. 따라서 짧은 이진수로 변환되는 (빈도가 높은) 원소는 허프만 나

무의 뿌리에서 가까운 잎에 위치하며, 긴 이진수로 변환되는 (빈도가 낮은) 원소는 허프만 나무의 뿌리에서 먼 잎에 위치한다.

허프만 나무는 각각의 원소에 대해서 잎으로만 이루어진 나무를 생성한 뒤 이들 나무를 하나의 나무로 합쳐가는 방식으로 만든다. 이 때 각 나무의 뿌리에는 원소들 빈도의 합을 저장한다.

1. 각각의 원소에 대해서 잎으로만 이루어진 나무를 생성한다. 잎에는 원소와 원소의 빈도를 저장한다.
2. 가장 낮은 빈도를 가지는 나무 T_1과 T_2를 선택한다. T_1과 T_2의 빈도 합을 각각 N_1과 N_2라고 하자. T_1과 T_2을 삭제하고 T_1과 T_2를 결합한 다음 나무를 새로 추가한다.

$$\begin{array}{c} N_1 + N_2 \\ \diagup \quad \diagdown \\ T_1 \quad T_2 \end{array}$$

   즉 새로 추가된 나무는 원소들 빈도의 합으로서 $N_1 + N_2$을 저장한다.
3. 하나의 나무만 남을 때까지 위 과정을 반복한다.

이렇게 하면 빈도가 낮은 원소일수록 일찍 2번 과정에서 이용되므로 최종적으로 만들어진 허프만 나무에서는 뿌리에서 먼 잎에 위치하게 된다.

허프만 나무의 생성 예

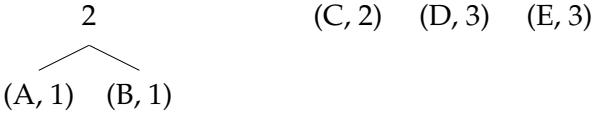
문자 A, B, C, D, E로 이루어진 정보를 압축하기 위해서 허프만 나무를 만든다고 하자. 문자들의 상대적 빈도는 다음과 같다.

A : 1, B : 1, C : 2, D : 3, E : 3

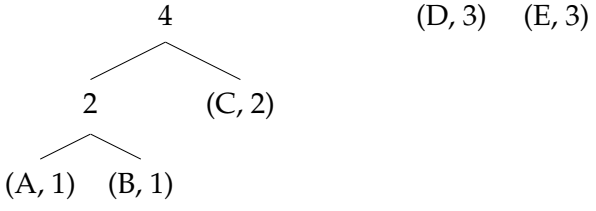
우선 하나의 잎으로만 이루어진 나무를 각각의 문자에 대해서 만든다:

(A, 1) (B, 1) (C, 2) (D, 3) (E, 3)

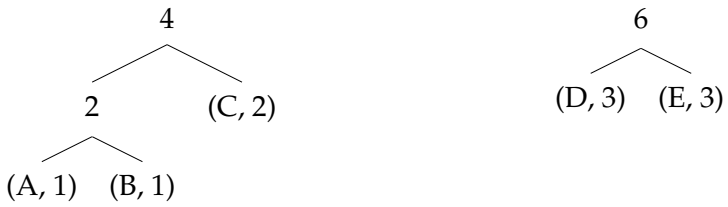
빈도가 가장 낮은 두 나무는 (A, 1)과 (B, 1)이므로 이 두 나무를 결합한다:



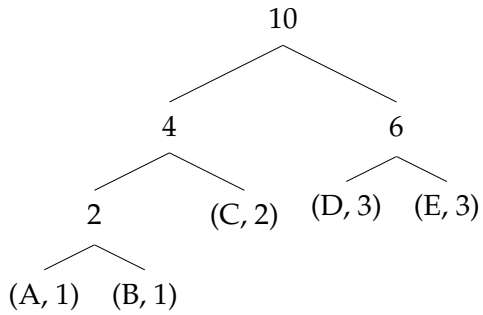
남은 4개의 나무 중 빈도가 가장 낮은 나무는 새로 생성된 나무와 (C, 2)이므로 이 두 나무를 결합한다:



남은 3개의 나무 중 빈도가 가장 낮은 나무는 처음에 만들어진 (D, 3)와 (E, 3)이므로 이 두 나무를 결합한다:



남은 두 나무를 결합하면 최종적으로 허프만 나무가 완성된다:



이 허프만 나무로부터 다음의 허프만 코드가 생성된다.

$$A \to 000, \quad B \to 001, \quad C \to 01, \quad D \to 10, \quad E \to 11$$