스칼라 튜토리얼

자바 프로그래머를 위하여

Version 1.3

2011년 3월 24일

Michel Schinz, Philipp Haller

역자: 이희종

(heejong@gmail.com)

PROGRAMMING METHODS LABORATORY

EPFL

SWITZERLAND

1 시작하면서

이 문서는 Scala 언어와 그 컴파일러에 대해 간단히 소개한다. 어느 정도의 프로그래밍 경험이 있으며 Scala 를 통해 무엇을 할 수 있는지를 빠르게 배우고 싶은 사람들을 위해 만들어 졌다. 여기서는 독자가 객체 지향 프로그래밍, 특히 Java 에 대한 지식을 가지고 있다고 가정한다.

2 첫 번째 예제

첫번째 예제로 흔히 쓰이는 *Hello world* 프로그램을 사용하자. 이 프로그램은 그다지 멋지지는 않지만 언어에 대한 많은 지식 없이도 Scala 언어를 다루는데 필요한 도구들의 사용법을 쉽게 보여 줄 수 있다. 아래를 보자:

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

자바 프로그래머들은 이 프로그램의 구조가 익숙 할 것이다. 프로그램은 문자열 배열 타입의 명령줄 인자를 받는 이름이 `main`인 함수 하나를 가지고 있다. 이 함수의 구현은 하나의 또 다른 함수 호출로 이루어져 있는데 미리 정의 된 함수 `println`에 어디선가 많이 본 바로 그 환영 메시지를 넘겨주어 호출 한다. `main` 함수는 값을 돌려주지 않기 때문에 리턴 타입을 선언 할 필요가 없다.

자바 프로그래머들에게 익숙하지 않은 부분은 `main` 함수를 감싸고 있는 `object` 선언일 것이다. 이 선언은 **싱글턴 객체**를 생성하는데, 이는 하나의 인스턴스만을 가지는 클래스라 할 수 있다. 따라서 위의 선언은 `HelloWorld` 라는 클래스와 역시 `HelloWorld` 라고 이름 붙인 이 클래스의 인스턴스를 함께 정의 하는 것이다. 이 인스턴스는 처음 사용 될 때에 필요에 따라 만들어 진다.

똑똑한 독자들은 이미 눈치챘겠지만 위의 예제에서 `main` 함수는 `static`이 아니다. Scala 에는 정적 멤버(함수든 필드든)라는 개념이 아얘 존재하지 않는다. 클래스의 일부로 정적 멤버를 정의하는 대신에 Scala 프로그래머들은 정적이기 원하는 멤버들을 싱글턴 객체안에 선언한다.

2.1 예제를 컴파일 하기

예제를 컴파일 하기 위하여 Scala 컴파일러인 `scalac` 를 사용한다. `scalac` 는 대부분의 컴파일러들과 비슷하게 동작한다. 소스파일과 필요에 따라 몇개의 옵션들을 인자로 받아 한개 또는 여러개의 오브젝트 파일을 생성한다. `scalac` 가 생성하는 오브젝트 파일은 표준적인 Java 클래스 파일이다.

위의 예제 프로그램을 `HelloWorld.scala`라는 이름으로 저장했다면, 아래의 명령으로 컴파일 할 수 있다 (부등호 '>'는 쉘 프롬프트이므로 함께 입력하지 말것) :

```
> scalac HelloWorld.scala
```

이제 현재 디렉토리에 몇개의 클래스 파일이 생성되는 것을 확인 할 수 있다. 그 중에 하나는 `HelloWorld.class`이며 `scala` 명령을 통해 바로 실행 가능한 클래스를 포함하고 있다. 다음 장을 보자.

2.2 예제를 실행하기

일단 컴파일 되면 Scala 프로그램은 `scala` 명령을 통해 실행 할 수 있다. 사용법은 Java 프로그램을 실행 할 때 사용하는 `java` 명령과 매우 비슷하며 동일한 옵션을 사용 가능하다. 위의 예제는 아래의 명령으로 실행 할 수 있으며 예상한대로의 결과가 나온다.

```
> scala -classpath . HelloWorld

Hello, world!
```

3 자바와 함께 사용하기

Scala 의 장점 중 하나는 Java 코드와 함께 사용하기 쉽다는 것이다. 사용하고 싶은 Java 클래스를 간단히 임포트 하면 되며, `java.lang` 패키지의 모든 클래스는 임포트 하지 않아도 기본적으로 사용 할 수 있다.

아래는 Scala 가 Java 와 얼마나 잘 어울리는지를 보여주는 예제이다. 우리는 아래 예제에서 현재의 날짜를 구하여 특정 국가에서 사용하는 형식으로 변환 할 것이다. 이를테면 프랑스[1]라 하자.

Java 의 클래스 라이브러리는 `Date`와 `DateFormat`과 같은 강력한 유틸리티 클래스를 가지고 있다. Scala 는 Java 와 자연스럽게 서로를 호출 할 수 있으므로, 동일한 역할을 하는 Scala 클래스 라이브러리를 구현하기 보다는 우리가 원하는 기능을 가진 Java 패키지를 간단히 임포트하여 이용하자.

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
}
```

Scala 의 임포트 구문은 Java 의 그것과 매우 비슷해 보이지만 사실 좀 더 강력하다. 위 예제의 첫번째 줄과 같이 중괄호를 사용하면 같은 패키지에서 여러개의 클래스를 선택적으로 불러 올 수 있다. Scala 임포트 구문의 또 한가지 특징은 패키지나 클래스에

[1] 불어를 사용하는 스위스의 일부 지역도 동일한 형식을 사용한다

속한 모든 이름들을 불러 올 경우 별표(*) 대신 밑줄(_) 을 사용 한다는 것이다. 별표는 Scala 에서 합법적인 식별자(함수명 등에 사용 가능한)로 사용된다. 나중에 자세히 살펴 볼 것이다.

따라서 세번째 줄의 임포트 구문은 `DateFormat` 클래스의 모든 멤버를 불러온다. 이렇게 함으로써 정적 함수 `getDateInstance` 와 정적 필드 `LONG`이 바로 사용 가능하게 된다.

`main` 함수 안에서 처음 하는 일은 Java 라이브러리에 속한 `Date` 클래스의 인스턴스를 생성하는 것이다. 이 인스턴스는 기본적으로 현재의 날짜를 가지고 있다. 다음으로 이전에 불러온 정적 함수 `getDateInstance` 를 통해 날짜 형식을 결정하고, 프랑스에 맞춰진 `DateFormat` 인스턴스를 사용하여 현재의 날짜를 출력한다. 이 마지막 줄은 Scala 문법의 재미있는 특성을 보여준다. 오직 하나의 인자를 갖는 함수는 마치 이항 연산자 같은 문법으로 호출 가능하다. 이 이야기는 곧 아래의 표현식이:

```
df format now
```

아래 표현식과 동일한 의미를 가진 다는 것이다. 그저 좀 더 간단하게 표현 되었을 뿐이다.

```
df.format(now)
```

이러한 특성은 그저 별것 아닌 문법의 일부 인것 처럼 보이지만 여러 곳에서 중요하게 사용 된다. 그중에 하나가 다음 장에 나와있다.

이번 장에서는 Java 와 Scala 가 얼마나 자연스럽게 서로 녹아드는지에 대해 배웠다. 이번 장에는 나타나지 않았지만, Scala 안에서 Java 의 클래스들을 상속받고 Java 의 인터페이스들을 바로 구현하는 것도 가능하다.

4 모든 것은 객체다

Scala 는 순수한 객체지향적 언어이다. 이 말은 곧 숫자와 함수를 포함한 **모든것**이 객체라는 것이다. 이러한 면에서 Scala 는 Java 와 다르다. Java 에서는 기본적인 타입(`boolean`이나 `int` 따위)과 참조 가능한 타입이 분리되어 있으며, 함수를 값과 동일하게 다룰 수도 없다.

4.1 숫자도 하나의 객체다

숫자는 객체이기 때문에 함수들을 포함하고 있다. 사실 아래와 같은 표현식은:

```
1 + 2 * 3 / x
```

오직 함수 호출로만 이루어져 있다. 우리가 이전 장에서 보았듯이, 위의 표현식은 아래의 표현식과 동일하다.

```
(1).+(((2).*(3))./(x))
```

위의 표현식처럼 +, * 등은 Scala 에서 합법적인 식별자이다.

위의 두번째 표현식에서 괄호는 꼭 필요하다. 왜냐하면 스칼라의 렉서(lexer)는 토큰들에 대하여 가장 긴 부분을 찾는 방법을 사용하기 때문이다. 아래의 표현식은:

```
1.+(2)
```

세개(1., +, 2)의 토큰들로 분리된다. 이렇게 토큰들이 분리되는 이유는 미리 정의되어 있는 유효한 토큰 중에 1. 이 1보다 길기 때문이다. 토큰 1. 은 리터럴 1.0으로 해석되어 Double 타입이 된다. 실제로 우리는 Int 타입을 의도 했음에도 말이다. 표현식을 아래와 같이 쓰면:

```
(1).+(2)
```

토큰 1이 Double로 해석 되는 것을 방지 할 수 있다.

4.2 함수마저 객체다

Java 프로그래머들에게는 놀라운 일이겠지만 Scala 에서는 함수도 역시 객체이다. 따라서 함수에 함수를 인자로 넘기거나, 함수를 변수에 저장하거나, 함수가 함수를 리턴하는 것도 가능하다. 이처럼 함수를 값과 동일하게 다루는 것은 매우 흥미로운 프로그래밍 패러다임인 **함수형 프로그래밍**의 핵심 요소 중 하나이다.

함수를 값과 같이 다루는 것이 유용함을 보이기 위해 아주 간단한 예제를 든다. 어떠한 행동을 매초 수행하는 타이머 함수를 생각해 보자. 수행 할 행동을 어떻게 넘겨 주어야 할까? 논리적으로 생각한다면 함수를 넘겨 주어야 한다. 함수를 전달하는 이런 종류의 상황은 많은 프로그래머들에게 익숙 할 것이다. 바로 유저 인터페이스 코드에서 어떤 이벤트가 발생하였을 때 불릴 콜백 함수를 등록하는 것 말이다.

아래 프로그램에서 타이머 함수의 이름은 oncePerSecond 이다. 이 함수는 콜백 함수를 인자로 받는다. 인자로 받는 함수의 타입은 () => Unit 인데, 이 타입은 인자를 받지 않고 아무 것도 돌려주지 않는 모든 함수를 뜻한다(Unit 타입은 C/C++에서 void 와 비슷하다). 이 프로그램의 메인 함수는 이 타이머 함수를 화면에 문장을 출력하는 간단한 콜백함수를 인자로 호출한다. 결국 이 프로그램이 하는 일은 일초에 한번씩 “time flies like an arrow”를 화면에 출력하는 것이 된다.

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

우리는 문자열을 화면에 출력하기 위하여 Scala 에 정의된 `println`을 사용 하였다. 이 함수는 Java 에서 흔히 사용하는 `System.out`에 정의된 것과 다르다.

4.2.1 이름없는 함수

이 프로그램은 이해하기 쉽지만 조금 더 다듬을 수도 있다. 함수 `timeFlies`는 오직 함수 `oncePerSecond`에 인자로 넘겨지기 위해 정의 되었다는 것에 주목하자. 이러한 한번만 사용되는 함수에 이름을 붙여 준다는 것은 필요 없는 일일 수 있다. 더 행복한 방법은 `oncePerSecond`에 함수가 전달 되는 그 순간 이 함수를 생성하는 것이다. Scala 에서 제공하는 **무명함수**를 사용하면 된다. 무명함수란 말 그대로 이름이 없는 함수이다. 함수 `timeFlies` 대신에 무명함수를 사용한 새로운 버전의 타이머 프로그램은 아래와 같다:

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

`main` 함수 안에 오른쪽 화살표 '=>'가 있는 곳이 무명함수이다. 오른쪽 화살표는 함수의 인자와 함수의 내용을 분리 해주는 역할을 한다. 위 예제에서 인자의 리스트는 비어있다. 화살표의 왼쪽을 보면 빈 괄호를 볼 수 있다. 함수의 내용은 `timeFlies`와 일치한다.

5 클래스에 대하여

지금까지 보았듯 Scala 는 객체지향적 언어이며 클래스의 개념이 존재한다.[2] Scala 의 클래스 정의는 Java 의 클래스 정의와 유사하다. 한가지 중요한 차이점은 Scala 클래스의 경우 파라미터들을 가질 수 있다는 것인데 아래 복소수 예제에 잘 나타나 있다:

```
class Complex(real: Double, imaginary: Double) {
  def re() = real
  def im() = imaginary
}
```

이 복소수 클래스는 두개의 인자를 받는다. 하나는 복소수의 실수 부분이고 다른 하나는 복소수의 허수 부분에 해당하는 값이 된다. 이 인자들은 `Complex` 클래스의 인스턴스를 생성 할 때 이처럼 반드시 전달 되어야 한다: `new Complex(1.5, 2.3)`. 클래스는 `re`와 `im`라는 두 함수를 가지고 있는데 각각의 함수를 통해 복소수를 구성하는 해당 부분의 값을 얻을 수 있다.

[2] 어떤 객체지향 언어는 클래스의 개념이 존재하지 않는다. 당연하게도 Scala 는 이들에 속하지 않는다.

이 두 함수의 리턴타입은 명시적으로 나타나 있지 않다는 사실에 주목하자. 컴파일러는 이 함수들의 오른편을 보고 둘 다 Double 타입을 리턴 한다고 자동으로 유추해 낸다.

하지만 컴파일러가 언제나 이렇게 타입을 유추해 낼 수 있는 것은 아니다. 그리고 불행하게도 어떤 경우 이러한 타입 유추가 가능하고 어떤 경우 불가능 한지에 관한 명확한 규칙도 존재하지 않는다. 일반적으로 이러한 상황은 별 문제가 되지 않는다. 왜냐하면 명시적으로 주어지지 않은 타입정보를 컴파일러가 자동으로 유추 해 낼 수 없는 경우 컴파일 시 에러가 발생하기 때문이다. 초보 Scala 프로그래머들을 위한 한가지 방법은, 주변을 보고 쉽게 타입을 유추 해 낼 수 있는 경우 일단 타입 선언을 생략하고 컴파일러가 받아 들이는지 확인하는 것이다. 이렇게 몇번을 반복하고 나면 프로그래머는 언제 타입을 생략해도 되고 언제 명시적으로 써주어야 하는지 감을 잡게 된다.

5.1 인자 없는 함수

함수 re와 im의 사소한 문제는 그들을 호출하기 위해 항상 뒤에 빈 괄호를 붙여 주어야 한다는 것이다. 아래를 보자:

```
object ComplexNumbers {
  def main(args: Array[String]) {
    val c = new Complex(1.2, 3.4)
    println("imaginary part: " + c.im())
  }
}
```

실수 부분과 허수 부분에 접근 할 때에 마치 그들이 필드인 것 처럼 함수 마지막에 빈 괄호를 붙이지 않을 수 있다면 더욱 좋겠다. 놀라지 마시라, Scala 는 이러한 기능을 완벽하게 제공한다. 그저 **인자를 제외**하고 함수를 정의하면 된다. 이런 종류의 함수는 인자가 0개인 함수와는 다른데, 인자가 0개인 함수는 빈 괄호가 따라 붙는 반면 이 함수는 정의 할 때도 사용 할 때도 이름 뒤에 괄호를 붙이지 않는다. 우리가 앞서 정의한 Complex 클래스는 아래와 같이 다시 쓸 수 있다:

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
}
```

5.2 상속과 재정의

모든 Scala 의 클래스들은 항상 상위 클래스로부터 상속된다. 만약 Complex 예제 처럼 상위 클래스가 존재하지 않을 경우는 묵시적으로 scala.AnyRef 를 상속한다.

Scala 에서는 물론 상위 클래스에 정의된 함수를 오버라이드 하는 것도 가능하다. 그러나 의도하지 않는 실수를 방지하기 위하여 다른 함수를 오버라이드 하는 함수는 override 지시자를 꼭 적어주어야 한다. 예를 들면, 우리의 Complex 클래스에 대해 Object 로 부터 상속된 toString 함수를 재정의 하는 법은 아래와 같다:

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
  override def toString() =
    "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

6 케이스 클래스 그리고 패턴 매칭

프로그램에 자주 등장하는 데이터 구조 중의 하나는 트리이다. 인터프리터와 컴파일러는 흔히 트리를 사용하여 내부 표현을 저장하고, XML 문서도 트리이며, 레드블랙 트리와 같은 저장구조 들도 트리에 기반을 두고 있다.

작은 계산기 프로그램을 통해 Scala 에서 이러한 트리들을 어떻게 표현하고 다루는지에 대해 알아 보자. 이 프로그램의 목표는 더하기와 상수인 정수 그리고 변수로 이루어진 간단한 산술 표현식을 다루는 것이다. 예를 들면, $1+2$나 $(x+x)+(7+y)$ 같은 식들 말이다.

처음으로, 우리는 해당 산술 표현식들을 어떻게 표현 할지 결정해야 한다. 가장 자연스러운 방법은 트리를 사용하는 것이다. 노드는 연산(여기서는 덧셈)이 될 것이고, 리프는 값(여기서는 상수 또는 변수)가 되겠다.

Java 였다면 트리를 나타내기 위해, 트리에 대한 추상 상위 클래스와 노드와 리프 각각에 대한 실제 하위 클래스들을 정의 했을 것이다. 함수형 언어였다면 같은 목적으로 대수적 데이터 타입을 사용 했을 것이다. Scala 는 **케이스 클래스**라 하는 이 둘 사이의 어디쯤에 놓여 질 수 있는 장치를 제공한다. 우리 예제의 트리 타입을 정의하기 위해 이 장치가 어떻게 사용 되는지 아래에서 실제적인 예를 보자:

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

클래스 `Sum`, `Var` 그리고 `Const`가 케이스 클래스로 선언되었다는 것은 이들이 여러 가지 면에서 일반적인 클래스와 다르다는 의미이다:

- 인스턴스를 생성 할 때 `new` 키워드를 생략 할 수 있다. 다른 말로, `new Const(5)` 라 쓰는 대신 `Const(5)` 라 쓰면 된다.
- 생성자 파라미터들에 대한 getter 함수가 자동으로 정의된다. 다른 말로, 클래스 `Const`의 인스턴스 `c`에 있는 생성자 파라미터 `v`의 값은 `c.v`로 접근 가능하다.
- 함수 `equals`와 `hashCode`도 공짜로 제공된다. 이 함수들은 레퍼런스의 동일함보다 **구조**의 동일함을 확인 하도록 구현되어 있다. 다른 말로, 생성 된 곳이 다르더라도 각각의 생성자 파라미터 값이 같다면 같은 것으로 여긴다.
- 함수 `toString`에 대한 기본적 구현이 제공된다. 이 기본적인 구현은 값이 생성될 때의 형태를 출력한다. 예를 들어 $x+1$의 트리 표현 을 출력 한다면 `Sum(Var(x),Const(1))`

이 된다.

- 케이스 클래스들의 인스턴스는 **패턴 매칭**을 통해 따로 사용 될 수 있다. 자세한 내용은 아래에서 다룬다.

산술 표현식을 나타낼 수 있는 데이터 타입을 정의 했으므로 이제 그것들을 계산 할 연산자들을 정의 할 차례다. 일단, 어떤 **환경**안에서 표현식을 계산 해주는 함수부터 시작하자. 환경은 각각의 변수마다 주어진 값들을 저장 해 두는 곳이다. 컴퓨터에서 메모리의 역할과 비슷 하다고 생각하면 된다. 예를 들어, 변수 x에 5가 저장된 환경 ($\{x \rightarrow 5\}$)에서 표현식 $x+1$을 계산하면 결과로 6이 나온다.

환경은 어떻게 표현하는게 좋을까? 간단히 생각하면, 해쉬 테이블 같은 두 값을 묶어주는 데이터 구조를 사용 할 수 있겠다. 그러나 우리는 이러한 데이터를 저장하는 목적으로 함수를 직접 사용 할 수도 있다! 가만 생각해 보면 환경이라는 것은 변수명에서 값으로 가는 함수에 지나지 않는다. 위에서 사용한 환경 $\{x \rightarrow 5\}$ 은 Scala 로 간단히 아래와 같이 쓴다:

```
{ case "x" => 5 }
```

이 문법은 함수를 정의한다. 이 함수는 문자열 `"x"`가 인자로 들어 왔을 때 정수 5를 돌려주고, 다른 모든 경우에 예외를 발생시키는 함수이다.

계산하는 함수를 작성하기 전에 환경 타입에 이름을 붙여 주는 것이 좋겠다. 물론 항상 환경 타입으로 `String => Int`를 사용해도 되지만 보기 좋은 이름을 붙이는 것은 프로그램을 더 읽기에 명료하고 변경에 유연하게 해 준다. Scala 에서는 아래와 같이 할 수 있다:

```
type Environment = String => Int
```

이제부터 타입 `Environment`는 `String`에서 `Int`로 가는 함수 타입의 다른 이름이다.

지금부터 계산하는 함수를 정의하자. 개념으로 따지면 매우 간단하다: 두 표현식의 합은 각 표현식의 값을 구하여 더한 것이다. 변수의 값은 환경에서 바로 가져 올 수 있고, 상수의 값은 상수 자체이다. 이것을 Scala 로 나타내는 것은 어렵지 않다:

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

이 계산 함수는 트리 `t`에 대해 **패턴 매칭**을 수행함으로써 동작한다. 위의 함수 정의는 직관적으로도 이해하기 쉽다:

1. 처음으로 `t`가 `Sum`인지 확인한다. 만약 맞다면 왼쪽 서브트리를 새로운 변수 `l`에 오른쪽 서브트리를 새로운 변수 `r`에 할당 한다. 그리고 화살표를 따라 화살표의 오른편으로 계산을 이어 나간다. 화살표의 오른편에서는 화살표의 왼편에서 할당된 변수 `l`과 `r`을 사용 한다.

2. 첫번째 확인이 성공하지 못하면 트리는 `Sum`이 아니라는 이야기이다. 다음으로는 `t`가 `Var`인지 확인한다. 만약 맞다면 `Var` 노드 안에 포함된 이름을 변수 `n`에 할당한다. 그리고 화살표의 오른쪽으로 진행한다.
3. 두번째 확인 역시 실패하면 `t`는 `Sum`도 `Var`도 아니라는 뜻이다. 이제는 `Const`에 대해 확인 해본다. 만약 맞다면 `Const` 노드 안의 값을 변수 `v`에 할당하고 화살표의 오른쪽으로 진행한다.
4. 마지막으로 모든 확인이 실패하면 패턴 매칭이 실패 했음을 알리는 예외가 발생하게 된다. 이러한 상황은 확인 한 것 외에 `Tree`의 하위 클래스가 더 존재 할 경우 일어난다.

패턴 매칭의 기본적인 아이디어는 대상이 되는 값을 여러가지 관심있는 패턴에 대해 순서대로 맞춰 본 후, 맞는 것이 있으면 맞은 값 중 관심 있는 부분에 대해 새롭게 이름 붙이고, 그 이름 붙인 부분을 사용하는 어떠한 작업을 진행하는 것이다.

객체지향에 숙련된 프로그래머라면 왜 `eval`을 클래스 `Tree`와 그 하위 클래스에 대한 **멤버 함수**로 정의하지 않았는지 궁금 할 것이다. 사실 그렇게 할 수도 있었다. Scala 는 일반적인 클래스 처럼 케이스 클래스에 대해서도 함수 정의를 허용한다. 패턴 매칭을 사용하느냐 멤버 함수를 사용하느냐는 사용자의 취향에 달린 문제다. 하지만 확장성에 관해 시사하는 중요한 점이 있다:

- 멤버 함수를 사용하면 단지 `Tree`에 대한 하위 클래스를 새롭게 정의 함으로 새로운 노드를 추가하기 쉽다. 반면에 트리에 대한 새로운 연산을 추가하는 작업이 고되다. 새로운 연산을 추가하기 위해서는 `Tree`의 모든 하위 클래스를 변경해야 하기 때문이다.
- 패턴 매칭을 사용하면 상황이 반대가 된다. 새로운 노드를 추가하려면 트리에 대해 패턴 매칭을 수행하는 모든 함수들을 새로운 노드도 고려하도록 변경해야 한다. 반면에 새로운 연산을 추가하는 것은 쉽다. 그냥 새로운 독립적인 함수를 만들면 된다.

패턴 매칭에 대해 좀 더 알아보기 위해, 산술 표현식에 대한 또 다른 연산을 정의 해보자. 이번 연산은 심볼 추출이다. 트리에서 우리가 원하는 특정 변수만 1로 표시하는 일이다. 독자는 아래 규칙만 기억하면 된다:

1. 더하기 표현식에서의 심볼 추출은 좌변과 우변의 심볼을 추출하여 더한 것과 같다.
2. 변수 v에 대한 심볼 추출은 v가 우리가 추출하기 원하는 심볼과 관련이 있다면 1이 되고 그 외의 경우 0이 된다.
3. 상수에 대한 심볼 추출 값은 0이다.

이 규칙들은 거의 그대로 Scala 코드가 된다.

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

위의 함수는 패턴 매칭에 관한 두 가지 새로운 기능을 소개한다. 첫 번째로, case 표현은 **가드**를 가질 수 있다. 가드란 if 키워드 뒤에 오는 표현식을 뜻하는 말로 패턴 매칭에 추가적인 조건을 부여한다. 가드가 참이 되지 않으면 패턴 매칭은 성공하지 못한다. 여기서는, 매칭 된 변수의 이름이 우리가 추출하는 심볼 v와 같을 때만 상수 1을 리턴함을 보장하는 용도로 사용된다. 두 번째 새로운 기능은 **와일드카드**이다. 밑줄 문자 _로 쓰며, 모든 값과 매치 되고 따로 이름을 붙이지 않는다.

매턴 매칭의 뛰어난 기능들을 모두 살펴보지는 못했지만, 문서를 너무 지루하게 만들지 않기 위하여 이쯤에서 멈추기로 한다. 이제 위에서 정의한 두 개의 예제 함수가 실제로 동작하는 모습을 보자. 산술 표현식 $(x+x)+(7+y)$에 대해 몇가지의 연산을 실행하는 간단한 main 함수를 만들기로 한다. 첫번째로 환경 $\{x \to 5, y \to 7\}$에서 그 값을 계산 할 것이고, 다음으로 x와 y에 대한 심볼 추출을 수행 할 것이다.

```
def main(args: Array[String]) {
  val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
  val env: Environment = { case "x" => 5 case "y" => 7 }
  println("Expression: " + exp)
  println("Evaluation with x=5, y=7: " + eval(exp, env))
  println("Derivative relative to x:\n " + derive(exp, "x"))
  println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

이 프로그램을 실행하면, 예상된 결과를 얻을 수 있다:

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

출력을 살펴 보면 심볼 추출의 결과가 사용자에게 좀 복잡하다는 생각이 든다. 패턴 매칭을 사용하여 이 결과를 단순화 하는 함수를 정의하는 것은 재미있는 문제이다 (생각보다 복잡하기도 하다). 독자들에게 연습문제로 남겨두겠다.

7 트레잇에 대하여

Scala 클래스에서는 상위 클래스에서 코드를 상속 받는 것 뿐만이 아니라, 하나 또는 여러개의 **트레잇(trait)**에서 코드를 불러 올 수 있는 방법도 있다.

Java 프로그래머들이 트레잇을 이해하는 가장 쉬운 길은 코드를 가질 수 있는 인터

페이스라고 생각하는 것이다. Scala 에서 어떤 클래스가 트레잇을 상속하면, 그 클래스는 트레잇의 인터페이스를 구현해야만 하고 동시에 트레잇이 가진 모든 코드들을 가져오게 된다.

트레잇의 유용함을 보이기 위해 객체들에 순서를 붙이는 고전적인 예제 하나를 들어보기로 하자. 순서가 있는 객체들은 정렬문제 처럼 주로 그들 사이에 비교가 필요할 경우 유용하다. Java 에서는 비교가능한 객체들이 `Comparable` 인터페이스를 구현하게 된다. Scala 에서는 이 `Comparable`을 트레잇으로 정의하여 더 나은 프로그램 디자인을 제공 할 수 있다. 여기서는 이를 `Ord`라 부를 것이다.

객체를 비교 할 때, 여섯개의 서로 다른 관계가 주로 사용 된다: 작다, 작거나 같다, 같다, 같지 않다, 크거나 같다, 크다. 하지만 이 여섯개를 일일히 구현하는 것은 지루하고 의미 없는 일이 될 것이다. 게다가 이중 두 가지 관계만 정의 되어도 나머지 네가지 관계를 계산 할 수 있지 않은가. 예를 들어 같다와 작다만 결정 할 수 있어도 나머지 관계의 참 거짓을 쉽게 판단 할 수 있다. Scala 에서는 이러한 논리들을 트레잇의 정의 안에 우아하게 표현 해 낼 수 있다:

```
trait Ord {
  def < (that: Any): Boolean
  def <=(that: Any): Boolean = (this < that) || (this == that)
  def > (that: Any): Boolean = !(this <= that)
  def >=(that: Any): Boolean = !(this < that)
}
```

위의 정의는 Java 의 `Comparable` 인터페이스와 같은 역할을 하는 `Ord`라고 불리는 새로운 타입을 만든다. 이 새로운 타입에는 세가지의 관계식이 기본적으로 구현이 되어 있으며 이 구현은 모두 하나의 추상 함수를 사용하고 있다. 모든 객체에 대해 기본적으로 존재하는 같다와 같지 않다에 대한 관계식은 빠져 있다.

위에서 사용된 타입 `Any`는 Scala 의 최상위 타입이다. Java 의 `Object` 타입과 같으나, `Int`, `Float`과 같은 기본 타입의 상위 타입이라는 점에서 좀 더 일반화 된 버전이라 생각 할 수 있다.

객체를 비교 가능하게 만들기 위해 정의해야 할 것은 같다와 작다 뿐이다. 나머지는 위의 `Ord` 트레잇을 삽입하여 처리한다. 하나의 예로 그레고리력의 날짜를 나타내는 `Date` 클래스를 만들어 보자. 이 날짜는 정수인 날, 월, 년으로 구성 된다. 일단 아래처럼 만든다:

```
class Date(y: Int, m: Int, d: Int) extends Ord {
  def year = y
  def month = m
  def day = d

  override def toString(): String = year + "-" + month + "-" + day
```

여기서 중요한 부분은 클래스 이름과 파라미터 뒤에 따라오는 `extends Ord` 선언이다. 이 선언은 `Date` 클래스가 `Ord` 트레잇을 상속함을 뜻한다.

다음으로 Object 에서 상속된 equals 함수를 재정의 하여 각각의 일, 월, 년을 비교하여 같음을 올바르게 판단하도록 한다. equals 의 기본 정의는 쓸모가 없다. 왜냐하면 Java 와 같이 기본적인 equals 는 물리적 주소를 비교하기 때문이다. 최종적인 코드는 다음과 같다:

```
override def equals(that: Any): Boolean =
  that.isInstanceOf[Date] && {
    val o = that.asInstanceOf[Date]
    o.day == day && o.month == month && o.year == year
  }
```

이 함수는 미리 정의된 함수인 isInstanceOf 와 asInstanceOf 를 사용한다. 첫번째 isInstanceOf 는 Java 의 instanceof 연산자와 동일한 일을 한다. 함수가 호출 된 객체가 함수의 인자로 들어온 타입의 인스턴스이면 참을 리턴한다. 두번째 asInstanceOf 는 Java 의 캐스트 연산자와 동일하다. 호출 된 객체가 인자로 들어온 타입의 인스턴스이면 그렇게 여겨지도록 변환하고 아니라면 ClassCastException 을 발생시킨다.

아래 마지막으로 정의된 함수는 작음을 판단하는 함수이다. 여기서는 error 라는 또 다른 미리 정의된 함수가 쓰였는데, 이 함수는 주어진 에러 메시지와 함께 예외를 발생 시키는 역할을 한다.

```
def <(that: Any): Boolean = {
  if (!that.isInstanceOf[Date])
    error("cannot compare " + that + " and a Date")

  val o = that.asInstanceOf[Date]
  (year < o.year) ||
  (year == o.year && (month < o.month ||
                      (month == o.month && day < o.day)))
}
```

이걸로 Date 클래스의 정의가 완성되었다. 이 클래스의 인스턴스는 날짜로도 또는 비교가능한 어떤 객체로도 여겨질 수 있다. 이들은 위에서 언급한 여섯가지 비교연산을 모두 가지고 있는데, equals 와 <는 Date 클래스의 정의 안에 직접 구현되어 있고 나머지는 Ord 트레잇에서 상속 받은 것이다.

트레잇은 여기서 예로 든 경우 외에도 물론 다양하게 사용 될 수 있다. 하지만 다양한 경우들에 대하여 깊게 다루는 일은 이 문서의 범위 밖이다.

8 제네릭함

이 튜토리얼에서 다룰 Scala 의 마지막 특징은 제네릭함이다. Java 프로그래머들은 Java 의 제네릭 지원이 부족하기 때문에 발생한 여러가지 문제점들에 대해 잘 알고 있을 것이다. 이 문제점들은 Java 1.5에서 다뤄졌다.

제네릭함이란 코드를 타입에 대하여 파라미터화 할 수 있는 능력이다. 이해를 돕기 위해 하나의 예를 들어 보자. 연결 리스트 라이브러리를 작성하는 프로그래머는 리스트의 원소 타입을 도대체 무엇으로 해야 할지 고민에 빠지게 된다. 이 연결 리스트는

서로 다른 많은 상황에서 사용 될 수 있기 때문에 원소의 타입이 반드시 Int 또는 반드시 Double이 될 것이라 미리 결정하는 것은 불가능하다. 이렇게 결정해 두는 일은 완전히 임의적이며 라이브러리의 사용에 있어 필요 이상의 심한 제약으로 작용 한다.

Java 프로그래머는 어쩔 수 없이 Object를 사용하곤 한다. Object는 모든 객체의 상위 타입이기 때문이다. 하지만 이런 방법은 이상적이지 않다. int, long, float등과 같은 기본 타입에 대해 동작하지 않으며, 연결 리스트에서 원소를 가져 올 때마다 많은 동적 타입 캐스트들을 프로그래머가 직접 삽입해 주어야 하기 때문이다.

Scala 는 이 문제를 해결하기 위한 제네릭 클래스와 제네릭 함수를 지원한다. 예제로 함께 살펴보자. 예제는 레퍼런스라는 간단한 저장구조 클래스이다. 이 클래스는 비어 있거나 또는 어떤 타입의 객체를 가리키는 포인터가 된다.

```
class Reference[T] {
  private var contents: T = _

  def set(value: T) { contents = value }
  def get: T = contents
}
```

클래스 Reference는 타입 T에 대해 파라미터화 되어있다. 타입 T는 레퍼런스의 원소 타입이다. 이 타입은 클래스 내부 여러 곳에서 나타나는데, contents 변수의 타입으로, set 함수의 인자 타입으로, 그리고 get 함수의 리턴 타입으로 사용 된다.

위의 코드 샘플은 Scala 에서 필드 변수를 만드는 내용이므로 따로 설명이 필요 없다. 한가지 흥미로운 점이 있다면 변수의 초기값이 _로 주어져 있다는 것인데, 여기서 _는 기본값을 뜻한다. 기본값은 수 타입에 대해서 0, Boolean 타입에 대해서 false, Unit 타입에 대해 (), 그리고 모든 객체 타입에 대해 null이다.

Reference 클래스를 사용하려면 타입 파라미터 T에 대해 적당한 타입을 지정해 주어야 한다. 이 타입은 레퍼런스 안에 들어갈 원소의 타입이 된다. 예를 들어, 정수 값을 저장 할 수 있는 레퍼런스를 생성하고 사용하기 위해서는 다음과 같이 쓴다:

```
object IntegerReference {
  def main(args: Array[String]) {
    val cell = new Reference[Int]
    cell.set(13)
    println("Reference contains the half of " + (cell.get * 2))
  }
}
```

위 예제에서 보듯 get 함수의 리턴값을 정수처럼 사용하기 위해 따로 캐스팅이 필요하지 않다. 여기서 정의된 레퍼런스는 정수를 포함하도록 선언이 되어 있으므로 정수 외에 다른 것은 넣을 수 없다.

9 마치며

우리는 지금까지 Scala 언어의 간략한 소개와 몇가지의 예제를 살펴 보았다. 흥미가 생겼다면 *Scala By Example*도 함께 읽어보자. 더 수준 높고 다양한 예제를 만날 수 있다. 필요 할 때마다 *Scala Langauge Specification*을 참고하는 것도 좋다.