EPFL

“for” Syntax

Effective Programming in Scala

for Syntax

Iterating over collections or transforming collections is so common in programming that Scala supports a special syntax aiming at making such pieces of code more readable.

As an example, consider a program that computes a list of phone numbers that start with the country prefix “+41”, and the name of the contact they belong to:

```scala
val namesAndSwissNumbers: List[(String, String)] =
  contacts.flatMap { contact =>
    contact.phoneNumbers
      .filter(phoneNumber => phoneNumber.startsWith(”+41”))
      .map(phoneNumber => (contact.name, phoneNumber))
  }
```

Processing Contacts, Take 2

The same program can be expressed as a “`for` expression”:

```
val namesAndSwissNumbers: List[(String, String)] =
  for
    contact     <- contacts
    phoneNumber <- contact.phoneNumbers
    if phoneNumber.startsWith(”+41”)
  yield (contact.name, phoneNumber)
```

Syntax of For

A for-expression is of the form

```
for ( s ) yield e
```

where `s` is a sequence of *generators* and *filters*, and `e` is an expression whose value is returned by an iteration.

- A *generator* is of the form `p <- e`, where `p` is a pattern and `e` an expression whose value is a collection.
- A *filter* is of the form `if f` where `f` is a boolean expression.
- The sequence must start with a generator.
- If there are several generators in the sequence, the last generators vary faster than the first.

Instead of `( s )`, the sequence of generators and filters can be written on multiple lines without requiring semicolons.

Translation of For (1)

The Scala compiler translates for-expressions in terms of `map`, `flatMap` and a lazy variant of `filter`.

Here is the translation scheme used by the compiler

1. A simple `for`-expression

```
for x <- e1 yield e2
```

is translated to

```
e1.map(x => e2)
```

Translation of For (2)

2. A `for`-expression

```
for (x <- e1 if f; s) yield e2
```

where `f` is a filter and `s` is a (potentially empty) sequence of generators and filters, is translated to

```
for (x <- e1.withFilter(x => f); s) yield e2
```

(and the translation continues with the new expression)

You can think of `withFilter` as a variant of `filter` that does not produce an intermediate list, but instead filters the following `map` or `flatMap` function application.

Translation of For (3)

3. A for-expression

```
for (x <- e1; y <- e2; s) yield e3
```

where `s` is a (potentially empty) sequence of generators and filters, is translated into

```
e1.flatMap(x => for (y <- e2; s) yield e3)
```

(and the translation continues with the new expression)

Exercise

```
for
  x <- 2 to N
  y <- 2 to x
  if x % y == 0
yield (x, y)
```

The expression above expands to which of the following two expressions?

O

```
(2 to N).flatMap { x =>
  (2 to x)
    .withFilter(y => x % y == 0)
    .map(y => (x, y))
}
```

O

```
(2 to N).map { x =>
  (2 to x).flatMap { y =>
    if (x % y) == 0 then (x, y)
  }
}
```

Exercise

```
for
  x <- 2 to N
  y <- 2 to x
  if x % y == 0
yield (x, y)
```

The expression above expands to which of the following two expressions?

X

```
(2 to N).flatMap { x =>
  (2 to x)
    .withFilter(y => x % y == 0)
    .map(y => (x, y))
}
```

O

```
(2 to N).map { x =>
  (2 to x).flatMap { y =>
    if (x % y) == 0 then (x, y)
  }
}
```

Imperative Loops

Imperative loops also have a special syntax. The statement:

```
for x <- e1 do s
```

(Note the usage of `do` instead of `yield`)

Is translated to:

```
e1.foreach(x => s)
```

Summary

The `for` expressions and statements *desugar* to calls to collection operations.

They can make combinatorial search more readable.