Refactoring Scala Programs to Promote Functional Design Patterns

Namrata Malkani; Manas Thakur
Indian Institute of Technology Mandi

KEY POINTS

- **Functional Design Patterns** - Immutability, declarative style.
- **Scala** - Hybrid language with excellent functional features.
- **Idiomatic Programming** - Concise, less error-prone, easy to implement & understand – like the functional features of Scala.
- **Loops -> Common Sequence Methods**.
- Single line solution, no mutation, less errors, more readable.
- **Conditionals -> Pattern Match**.
- Better readability, enforces common return value, efficient byte-code.
- **ScalaRT**: Scala Refactoring Tool.
- Suggest improvements on code to implement idiomatic style.
- Easily extendable, can be imported in packages in Scala projects based on need.
- Stands at 1300+ lines of code.

FUTURE RESEARCH

- More refactoring, implementing currying, tackling larger OO patterns.
- Evaluating the original and refactored program with respect to a code-evaluation metric, thus coming up with a code-quality check system.

REFERENCES

[1] Michael Bevilacqua-Linn. *Functional Programming Patterns*. The Pragmatic Programmers, LLC.

[2] ScalaMeta.Org. Scalameta- Library to read, analyse, transform and generate Scala programs, 2016.

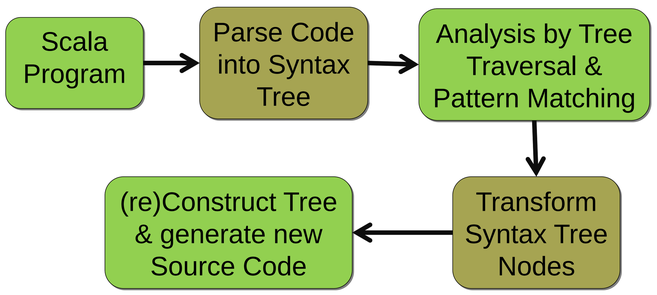
AUTHORS' CONTACT

Namrata Malkani
Indian Institute of Technology Mandi
Email: nmalkani13@gmail.com

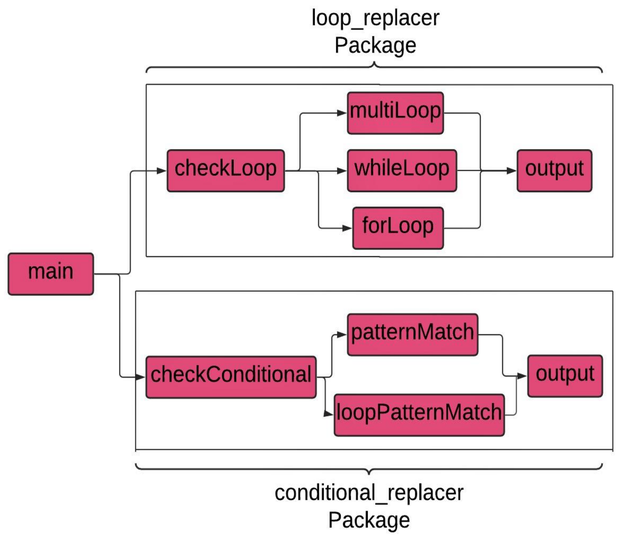
Manas Thakur
Indian Institute of Technology Mandi
Email: manas@iitmandi.ac.in

SCALA CODE ANALYSIS & REFACTORING

- **Refactoring**: Transforming program to improve its design, structure, and implementation, while preserving its functionality.
- It requires a framework that lets the user work on the underlying **syntax tree** of the source code. We use the metaprogramming library **Scalameta**.



ScalaRT: TOOL DESIGN



LOOPS TO COMMON SEQUENCE METHODS

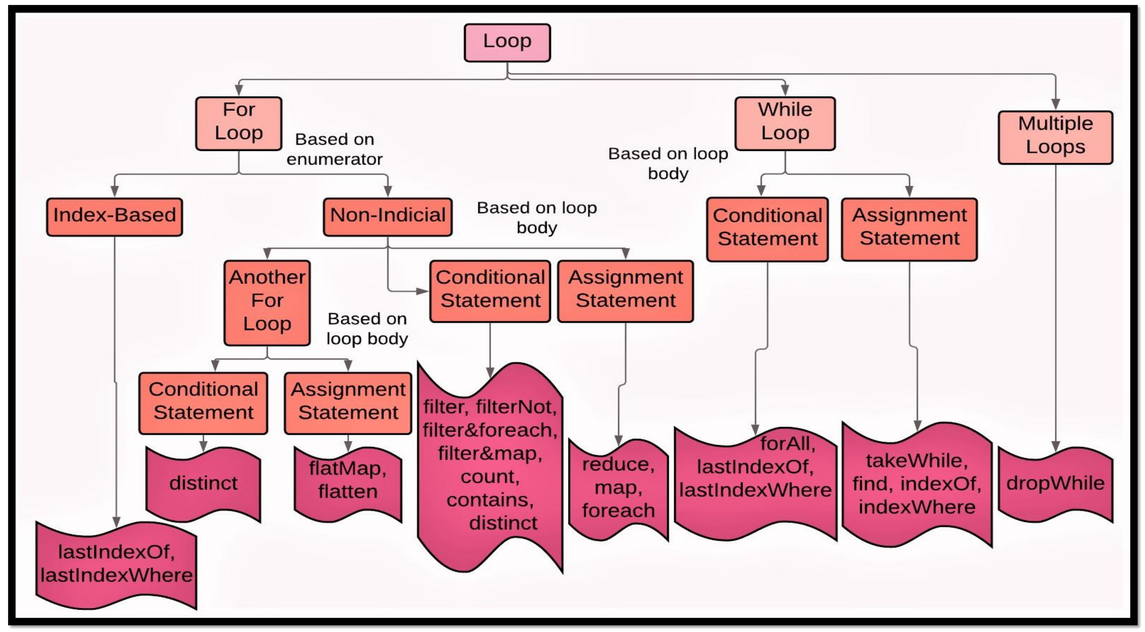


Figure 1. Loop Categorization. Further, assigning a sequence method to the loop is based on tree-node analysis of the key assignment or conditional statement.

Tool Output	Loop code snippet
xs.filter(h).map(x =⟩ g(x))	`for (x <- xs) { if (h(x)) list += g(x) }`
xs.filter(h)	`for (x <- xs) { if (h(x)) list += g(x) }`
xs.takeWhile(h)	`while (h(xs(i))) { list += xs(i) i +=1 }`
Likely sequence method- flatMap	`for(x <- xs){ for(t <- f(x)){ ans += t } }`

Table 1. Loop refactoring examples. Exact refactored code or likely sequence method replacement can be given from among 20 sequence methods in Scala 2.

CONDITIONALS TO PATTERN MATCH EXPRESSIONS

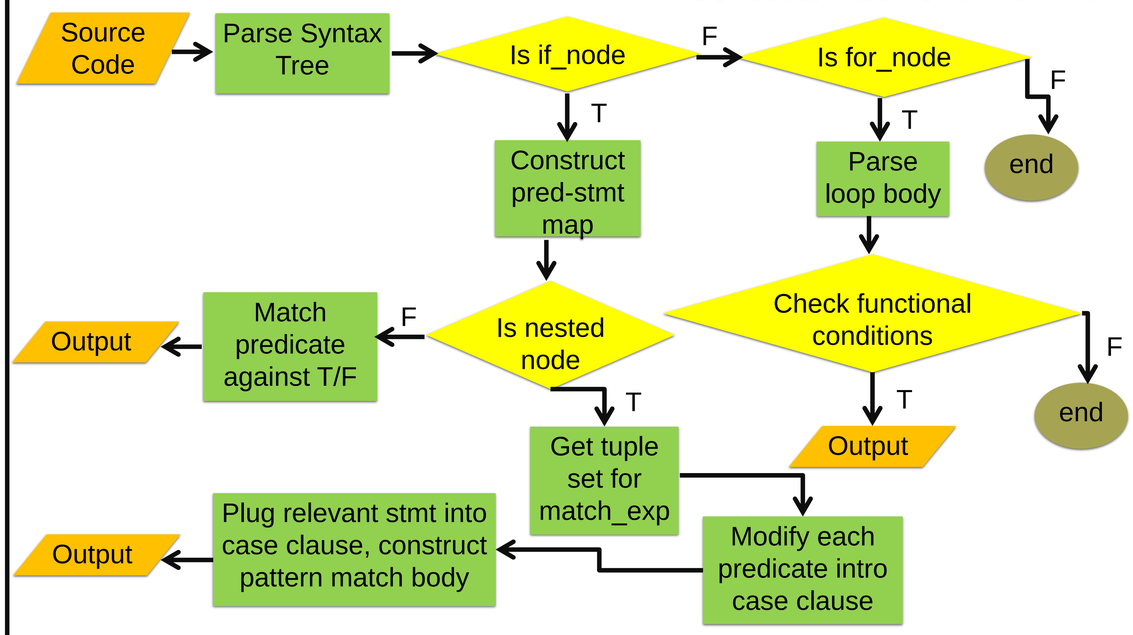


Figure 2. Algorithm for refactoring conditional statements. All sorts of if-else, nested else-if statements can be converted to pattern matching, if tuple size remains upto 3. Otherwise refactoring would likely reduce readability.

```
1  if (a == 0) f(a)
2  else if (b==10 && a==
       5)
3     f(a+b)
4  else a + b + c
```

```
1  for (x <- xs){
2    if(x==1)
3      result += f(x+1)
4    else if(x==2)
5      result += f(x+2)
6    else
7      result += f(x+4)
8  }
```

```
1  (a, b) match {
2  case (0,_) => f(a)
3  case (5,10) =>
4     f(a + b)
5  case (_,_) =>
6     a + b + c
7  }
```

```
1  result = xs.map{
2    x => x match {
3      case 1 => f(x+1)
4      case 2 => f(x+2)
5      case _ => f(x+4)
6    }
7  }
```

Figure 3. Conditional refactoring examples. A special case of conditional enclosed within loop- refactored to pattern matching function within 'map' is shown in the bottom.