Liberal Syntax

Compiler Construction '15 Final Report

Ireneu Pla Gabriel Luthier
EPFL
ireneu.pla@epfl.ch gabriel.luthier@epfl.ch

1. Introduction

We wrote a complete compiler. It starts by reading the code using the Lexer and decoding it as Tokens. These Tokens are parsed according to the grammar imposed by the Tool language. After this, the code is analysed and type checked. Finally, the compiler generates .class files so that the Tool program can run on the Java Virtual Machine (JVM).

Our project goal was to allow a more liberal syntax for the Tool programming language. Four main features where to be implemented.

1. **Semicolon inference**: optional semicolons to indicate the end of an instruction. To do this, our parser does the longest possible match on instructions. This means that it is possible to span an instruction over several lines and that it will be parsed correctly.
2. **Methods as infix operators**: ability to call methods in an infix way. This applies only to methods that have only one argument to keep the code readable. Methods with several arguments still have to be called using the standard Tool notation.
3. **Expressions as statements**: let expressions take the place of statements. Even though it is not very interesting to be able to have literals hanging around in the code, it allows the Tool programmer to call methods of an object without necessarily having to assign the return value to a variable. This can help to remove a lot of code clutter.
4. **Operators as methods**: override or define operators for custom classes. This feature makes it possible to have methods defined with operators (+, -, *...) for the user's classes. The aim of this extension is to allow the programmer to have clearer code by being able to define these methods for any class.

Note: *The first thing we had to do was to correct the mistakes in CodeGeneration to be able to output correct Java .class files.*

2. Examples

2.1 Semicolon inference

This extension allows us to write code without semicolons at the end of the lines. The semicolon becomes an optional element in a statement. It is however still necessary to separate multiple statements on the same line of code.

```
object SemiColonInference {
    def main() : Unit = {
        println(new Tester().test())
    }
}

class Tester {
    def test() : Int = {
                var a : Int
                var b : Int

                a = 1; b = 2

                a =
                        a
                        + b
                        * 4

                return a
    }
}
```

2.2 Methods as infix operators

This extension allows us to call methods not only in prefix form but also in an infix way. As stated before, only methods that have one argument can be called in this way.

```
object InfixMethods {
    def main() : Unit = {
        println(new Tester().test())
    }
}

class Tester {
    def test() : Int = {
                var myObject : Object
                myObject = new Object().init()

                return myObject add 6
    }
}

class Object {
        var value : Int

        def init() : Object = {
                value = 14
                return this
        }

        def add(a: Int) : Int = {
                return value + a
        }
}
```

2.3 Expressions as statements

This extension allow to use expressions where a statement was previously expected.

```
object InfixMethods {
    def main() : Unit = {
        println(new Tester().test())
    }
}

class Tester {
    def test() : Int = {
                var myObject : Object
                myObject = new Object().init()

                myObject resetTo 1

                return myObject add 6
    }
}

class Object {
        var value : Int

        def init() : Object = {
                value = 14
                return this
        }

        def resetTo(x: Int) : Int = {
            value = x
            return 0
        }
}
```

You can notice that the return value of myObject resetTo 1 was not assigned to any variable, but that it's action is still executed.

2.4 Operators as methods

This extension allow us to name methods with operators, such as ||, &&, <, ==, +, −, ∗, / or !.

```
object OperatorsAsMethods {
    def main() : Unit = {
        println(new Tester().test())
    }
}

class Tester {
    def test() : Int = {
                var a: ArbitraryObject
                var b: ArbitraryObject

                a = new ArbitraryObject().init()
                b = new ArbitraryObject().init()

                return a.+(b) // or a + b
    }
}

class ArbitraryObject {
        def init() : ArbitraryObject = {
                return this;
        }

        def +(b: ArbitraryObject): Int = {
                return 100
        }
}
```

3. Implementation

To implement all extensions, we modified the Parser.

3.1 Semicolon inference

Implementing this extension was quite straightforward. Every time the grammar stated that a semicolon was needed, we relaxed this rule. Now if a semicolon is

found at the old place, we eat it and continue. But we do the same thing if no semicolon is found.

3.2 Methods as infix operators

To implement this extension, we added the grammar rule: expression identifier expression.

3.3 Expressions as statements

3.4 Operators as methods

To implement this extension, we first modified the parsing of a method declaration. After the DEF token, we check if an operator was following. If it is the case, we eat it and add an Identifier corresponding to the operator. If it is not the case, we parse a simple Identifier. Then, for the method calls, we added some cases when we found a DOT in the code. If an operator was following, we interpreted it as a method call with the corresponding Identifier.

4. Possible Extensions

We were not able to implement calls to operators as methods in infix form.