Liberal Syntax

Compiler Construction '15 Final Report

Ireneu Pla Gabriel Luthier
EPFL
{firtname.lastname}@epfl.ch

1. Introduction

We wrote a complete compiler. It starts by reading the code and decoding it as Tokens. With the Tokens, it parse them by following the grammar imposed by the Tool language. Then it analyze it and attribute each Identifier a symbol in order to know which variable it is using. At the end, it checks the types of all Identifier and generate the associated code.

2. Examples

2.1 Semicolon inference

This extension allow us to write code without any semicolon at the end of the lines. We can still specify it if wanted. It is compulsory if we want to write multiple statements in one line.

```
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 extesion allow us to call methods not only in prefix form but also as infix operators. It is only allowed to use this form when methods have one argument.

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

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

                return myObject infix_method 6
    }
}

class Object {
        var value : Int

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

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

2.3 Expressions as statements

This extension allow to use expressions also when we previously expected a statement.

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)
    }
}

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 iin infix form.