SYNTAX AND SEMANTICS

BY:-
Ayush Garg - CS16B003
Pratik Agarwalla - CS16B020
Prudhvivenkat - CS16B022
Tauseef Ahmed - CS16B027

March 27, 2018

Contents

1 Introduction

Scala provides support for both functional programming and a strong static type system. It is a multi-paradigm language i.e. functional, object oriented, imperative and concurrent. In Scala, code sizes are reduced by a factor of two or three when compared to the Java. Scala programs can run on JVM so we can make use of Java libraries. Also, a programming language was needed which can be object-oriented programming language as well as functional language. So, Scala was integrated well with existing programming languages like Java and C#. **Scala scales with the programmer.**

- Scala as an object-oriented programming language, treats every value as an object and whatever operation you do in it is treated as a method call As a functional programming language it's very easy to define function here. We can use nested and recursive function here. We can add new language constructs here.
- XML data can be processed with the help of Scala. Scala is used for pattern matching for which it is used in wen development and stuff.
- Scala includes feature like operator overloading, named parameters, optical parameters and raw string.
- Function name can be a single unique symbol or a unique sequence of symbols. Any method can be used as a infix operator.
- All variables are preceded by var or val, we don't need to declare a particular data type of a variable.

2 Syntax and Coding Conventions

- Scala is a case sensitive language, which means 'Abc' and 'abc' will have different meaning.
- In Scala, class name starts with capital letter. If class name consists of more than one word, each word's first letter should be capital. Ex - class Abc
- In Scala, whenever you declare a method, it should start with small letter. Ex - def abc
- Program file name should be exactly equal to object name. If Xyz is the object name, it should be saved as Xyz. Scala
- Main () function is the mandatory part in Scala programs.
- Scala supports single line and multi-line comment. Whatever you write between the comments is completely ignored. // single line comment
  /* multi line comment*/

- Packages in Scala can be imported using import Scala.Library._
  For ex - import Scala.collection._

To construct a token characters are distinguished in following classes.

1. Whitespace characters. ŭ0020 — ŭ0009 — ŭ000D — ŭ000A
2. Letters, which include lower case letters, upper case letters
3. Digits '0' — . . . — '9'
4. Parentheses '(' — ')' — '[' — ']' — '' — ''
5. Delimiter characters "' — '" — '"' — '.' — ';' — ','.
6. Operator characters.

Identifiers in Scala are of 4 types:

1. **Alphanumeric identifies** can start with a letter or an underscore and are followed by letters and underscore. For ex: _xyz
2. Using one or more operators, we can declare a **operator identifier**
3. Combination of both operator identifier and alphanumeric identifier gives **mixed identifier**
4. We can declare a **literal identifier** enclosed in back ticks.

Variables: A variable which can change value is called mutable value and is declared with the help of keyword var while the variable which cannot change its value is called immutable value and is declared with the help of keyword val.
For ex. var abc, val xyz
We can also define data type of a variable using the following syntax:
var abc: datatype = value
Scala has variable type inference ,i.e., when we declare a variable with val or var without using data type it automatically recognises the data type based on the value of variable.

3 Basic Declarations and Definitions

3.1 Value Declarations and definitions

```
Syntax:
Dcl  ::= 'val' ValDcl
ValDcl  ::= ids ':' Type
PatVarDef  ::= 'val' PatDef
PatDef  ::= Pattern2 {',' Pattern2} [':' Type] '=' Expr
Ids  ::= id {',' id}
```

3.2 Variable Declarations and Definitions

```
Syntax:
Dcl  ::= ‘var’ VarDcl
PatVarDef  ::= ‘var’ VarDef
VarDcl  ::= ids ‘:’ Type
VarDef  ::= PatDef
     | ids ‘:’ Type ‘=’ ‘_’
```

A variable definition var x: T = _ can appear only as a member of a template. It introduces a mutable field with type T and a default initial value. The default value depends on the type T as follows:

```
0  if T is Int or one of its subrange types,
0L  if T is Long,
0.0f  if T is Float,
0.0d  if T is Double,
false  if T is Boolean,
()  if T is Unit,
null  for all other types T .
```

3.3 Type Parameters

```
Syntax:
TypeParamClause  ::= ‘[’ VariantTypeParam {‘,’ VariantTypeParam} ‘]’
VariantTypeParam  ::= {Annotation} [‘+’ | ‘’]TypeParam
TypeParam  ::= (id | ‘_’) [TypeParamClause] [‘>:’ Type] [‘<:’ Type] [‘:’ Type]
```

3.3.1 Function Declarations and Definitions

```
Syntax:
Dcl  ::= ‘def’ FunDcl
FunDcl  ::= FunSig ‘:’ Type
Def  ::= ‘def’ FunDef
FunDef  ::= FunSig [‘:’ Type] ‘=’ Expr
FunSig  ::= id [FunTypeParamClause] ParamClauses
FunTypeParamClause  ::= ‘[’ TypeParam {‘,’ TypeParam} ‘]’
ParamClauses  ::= {ParamClause} [[nl] ‘(’ ‘implicit’ Params ‘)’]
ParamClause  ::= [nl] ‘(’ [Params] ‘)’}
Params  ::= Param {‘,’ Param}
Param  ::= {Annotation} id [‘:’ ParamType] [‘=’ Expr]
ParamType  ::= Type
| ‘=>’ Type
| Type ‘*
```

4 Expressions

```
Syntax:
Expr ::= (Bindings | id | ‘_’) ‘=>’ Expr
                                  | Expr1
Expr1 ::= ‘if’ ‘(’ Expr ‘)’ {nl} Expr [[semi] else Expr]
        | ‘while’ ‘(’ Expr ‘)’ {nl} Expr
        | ‘try’ ‘{’ Block ‘}’ [‘catch’ ‘{’ CaseClauses ‘}’]
        [‘finally’ Expr]
        | ‘do’ Expr [semi] ‘while’ ‘(’ Expr ’)’
        | ‘for’ (‘(’ Enumerators ‘)’ | ‘{’ Enumerators ‘}’)
{nl} [‘yield’] Expr
        | ‘throw’ Expr
        | ‘return’ [Expr]
        | [SimpleExpr ‘.’] id ‘=’ Expr
        | SimpleExpr1 ArgumentExprs ‘=’ Expr
        | PostfixExpr
        | PostfixExpr Ascription
        | PostfixExpr ‘match’ ‘{’ CaseClauses ‘}’
PostfixExpr ::= InfixExpr [id [nl]]
InfixExpr ::= PrefixExpr
| InfixExpr id [nl] InfixExpr
PrefixExpr ::= [‘’| ‘+’ | ‘~’ | ‘!’] SimpleExpr
SimpleExpr ::= ‘new’ (ClassTemplate | TemplateBody)
             | BlockExpr
             | SimpleExpr1 [‘_’]
SimpleExpr1 ::= Literal
          | Path
  | ‘_’
              | ‘(’ [Exprs] ‘)’
              | SimpleExpr ‘.’ id s
              | SimpleExpr TypeArgs
              | SimpleExpr1 ArgumentExprs
              | XmlExpr
Exprs ::= Expr {‘,’ Expr}
BlockExpr ::= ‘{’ CaseClauses ‘}’
            | ‘{’ Block ‘}’
Block ::= {BlockStat semi} [ResultExpr]
ResultExpr ::= Expr1
| (Bindings | ([‘implicit’] id | ‘_’) ‘:’ CompoundType) ‘=>’ Block
Ascription ::= ‘:’ InfixType
             | ‘:’ Annotation {Annotation}
             | ‘:’ ‘_’ ‘*’
```

4.1 Literals

```
Syntax:
SimpleExpr ::= Literal
Typing of literals is as described; their evaluation is immediate.
```

4.2 Designators

```
Syntax:
SimpleExpr ::= Path
             | SimpleExpr ‘.’ Id
```

4.3 This and Super

```
Syntax:
SimpleExpr  ::= [id ‘.’] ‘this’
               | [id ’.’] ‘super’ [ClassQualifier] ‘.’ Id
```

4.4 Function Applications

```
SimpleExpr ::= SimpleExpr1 ArgumentExprs
ArgumentExprs ::= ‘(’ [Exprs] ‘)’
                 | ‘(’ [Exprs ‘,’] PostfixExpr ‘:’ ‘_’ ‘*’ ’)’
                 | [nl] BlockExpr
Exprs ::= Expr {‘,’ Expr}
```

4.5 Tuples

```
SimpleExpr ::= ‘(’ [Exprs] ‘)’
```

4.6 Instance Creation Expressions

```
Syntax:
SimpleExpr ::= ‘new’ (ClassTemplate | TemplateBody)
```

4.7 Blocks

```
Syntax:
BlockExpr ::= ‘{’ Block ‘}’
Block ::= {BlockStat semi} [ResultExpr]
```

4.8 Typed Expressions

```
Syntax:
Expr1 ::= PostfixExpr ‘:’ CompoundType
```

The typed expression e : T has type T . The type of expression e is expected to conform to T . The result of the expression is the value of e converted to type T.

4.9 Annotated Expressions

```
Syntax:
Expr1 ::= PostfixExpr ‘:’ Annotation {Annotation}
```

4.10 Assignments

```
Syntax:
Expr1 ::= [SimpleExpr ‘.’] id ‘=’ Expr
    | SimpleExpr1 ArgumentExprs ‘=’ Expr
```

4.11 Conditional Expressions

```
Syntax:
Expr1 ::= ‘if’ ‘(’ Expr ‘)’ {nl} Expr [[semi] ‘else’ Expr]
```

4.12 While Loop Expressions

```
Syntax:
Expr1 ::= ‘while’ ‘(’ Expr ’)’ {nl} Expr
def whileLoop(cond: => Boolean)(body: => Unit): Unit =
if (cond) { body ; whileLoop(cond)(body) } else {}
```

4.13 Do Loop Expressions

```
Syntax:
Expr1 ::= ‘do’ Expr [semi] ‘while’ ‘(’ Expr ’)’
```

4.14 For Comprehensions and For Loops

```
Syntax:
Expr1 ::= ‘for’ (‘(’ Enumerators ‘)’ | ‘{’ Enumerators ‘}’)
  {nl} [‘yield’] Expr
Enumerators ::= Generator {semi Enumerator}
Enumerator ::= Generator
             | Guard
             | Pattern1 ‘=’ Expr
Generator ::= Pattern1 ‘<-’Expr [Guard]
Guard ::= ‘if’ PostfixExpr
```

4.15 Return Expressions

```
Syntax:
Expr1 ::= ‘return’ [Expr]
```

4.16 Throw Expressions

```
Syntax:
Expr1 ::= ‘throw’ Expr
```

4.17 Try Expressions

```
Syntax:
Expr1 ::= ‘try’ ‘{’ Block ‘}’ [‘catch’ ‘{’ CaseClauses ‘}’]
[‘finally’ Expr]
```

4.18 Anonymous Functions

```
Syntax:
Expr ::= (Bindings | [‘implicit’] id | ‘_’) ‘=>’ Expr
ResultExpr ::= (Bindings | ([‘implicit’] id | ‘_’) ‘:’ CompoundType) ‘=>’ Block
Bindings ::= ‘(’ Binding {‘,’ Binding} ‘)’
Binding ::= (id | ‘_’) [‘:’ Type]
```

5 Implicit Parameters and Views

5.1 The Implicit Modifier

```
Syntax:
LocalModifier ::= ‘implicit’
ParamClauses ::= {ParamClause} [nl] ‘(’ ‘implicit’ Params ‘)’
```

5.2 Views

Implicit parameters and methods can also define implicit conversions called views. A view from type S to type T is defined by an implicit value which has function type S=¿T or (=¿S)=¿T or by a method convertible to a value of that type.
Views are applied in three situations.
1. If an expression e is of type T , and T does not conform to the expression’s expected type pt. In this case an implicit v is searched which is applicable to e and whose result type conforms to pt. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of T =¿ pt. If such a view is found, the expression e is converted to v(e).
2. In a selection e.m with e of type T , if the selectorm does not denote an accessible member of T . In this case, a view v is searched which is applicable to

e and whose result contains a member namedm. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of T . If such a view is found, the selection e.m is converted to v(e).m.
3. In a selection e.m(args) with e of type T , if the selectorm denotes some member(s) of T , but none of these members is applicable to the arguments args. In this case a view v is searched which is applicable to e and whose result contains a method m which is applicable to args. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of T . If such a view is found, the selection e.m is converted to v(e).m(args).

5.3 Context Bounds and View Bounds

```
Syntax:
TypeParam ::= (id | '_') [TypeParamClause] ['>:' Type] ['<:'Type]
{'<%' Type} {':' Type}
```

5.4 Manifests

Manifests are type descriptors that can be automatically generated by the Scala compiler as arguments to implicit parameters. The Scala standard library contains a hierarchy of four manifest classes, with OptManifest at the top. Their signatures follow the outline below.

```
trait OptManifest[+T]
object NoManifest extends OptManifest[Nothing]
trait ClassManifest[T] extends OptManifest[T]
trait Manifest[T] extends ClassManifest[T]
```

6 Classes And Objects

Classes and objects are defined in terms of templates in Scala.
Syntax:

```
TemplateDef ::= ['case'] 'class' ClassDef
              |[ 'case' ] 'object' ObjectDef
              |'trait' TraitDef
```

A template contains information about the type signature, behavior and initial state of a trait or class of objects or of a single object. Templates are an integral part of instance creation expressions and class and object definitions.

6.1 Class Linearization

The classes reachable through transitive closure of the direct inheritance relation from a class C, are called the base classes of C. This is called Class

Linearization.
Consider the following class definitions:

```
abstract class AbsIterator extends AnyRef { ... }
trait RichIterator extends AbsIterator { ... }
class StringIterator extends AbsIterator { ... }
class Iter extends StringIterator with RichIterator { ... }
```

Then the linearization of class Iter is

```
{ Iter, RichIterator, StringIterator, AbsIterator, AnyRef, Any }
```

Note that the linearization of a class refines the inheritance relation: if C is a subclass of D, then C precedes D in any linearization where both C and D occur.

6.2 Class Members

The class name works as a class constructor that can take a number of parameters. Class variables are called, fields of the class and methods declared in the class are called class methods. A class C can define members in its statement sequence and can inherit members from all parent classes. Objects of a class can be created using a keyword new and then the class fields and methods can be accessed using the newly declared object of that class using the '.' operator.

6.3 Inheritance Closure

Let C be a class type. The inheritance closure of C is the smallest set S of types such that if T is a class type in S, then all parents of T are also in S.

6.4 Early definitions

Early definitions are particularly useful for traits, which do not have normal constructor parameters. This shows the lazy feature of Scala.
Example:

```
trait Greeting {
val name: String
val msg = "How are you, "+name
}
class C extends {
val name = "Bob"
} with Greeting {
println (msg)
}
```

In the code above, the field name is initialized before the constructor of Greeting is called. Therefore, field msg in class Greeting is properly initialized to "How are you, Bob". However, if name had been initialized in C's normal class body, it would be initialized after the constructor of Greeting. In that case, msg would be initialized to "How are you, ¡null¿".

6.5 Modifiers

Modifiers, which affect the accessibility and usage of the identifiers bound by them, precede member definitions of a class. The rules governing the validity and meaning of a modifier are as follows.
• The private modifier can be used with any definition or declaration in a template. Such members can be accessed only from within the directly enclosing template. They are not inherited by subclasses and they may not override definitions in parent classes.
• The protected modifier applies to class member definitions. Protected members of a class can be accessed from within the template of the defining class, and all templates that have the defining class as a base class.
• The override modifier applies to class member definitions or declarations. It is mandatory for member definitions or declarations that override some other concrete member definition in a parent class.
• The final modifier applies to class member definitions and to class definitions. A final class member definition cannot be overridden in subclasses and a template cannot inherit a final class.

6.6 Class Definitions

Syntax:

```
TemplateDef::= ‘class’ ClassDef
ClassDef::= id [TypeParamClause] {Annotation}
            [ AccessModifier ] ClassParamClauses ClassTemplateOpt
ClassParamClauses::= {ClassParamClause}
                     [[nl] ‘(’ implicit ClassParams ‘)’]
ClassParamClause::= [nl] ‘(’ [ClassParams] ’)’
ClassParams::= ClassParam {‘,’ ClassParam}
ClassParam::= {Annotation} [{Modifier} (‘val’ | ‘var’)]
              id [‘:’ ParamType] [‘=’ Expr]
ClassTemplateOpt::= ‘extends’ ClassTemplate | [[‘extends’] TemplateBody]
```

The most general form of class definition is
class c[tps] as m(ps1). . .(psn) extends t (n ¸ 0).
Here,
• c is the name of the class to be defined.
• [tps] is a non-empty list of type parameters of the class being defined.
• as is a possibly empty sequence of annotations.

• m is an access modifier such as private or protected, possibly with a qualification.
• (ps1) . . . (psn) are formal value parameter of the class. The scope of a formal value parameter is the whole class c. If no formal parameter sections are given, an empty parameter section () is assumed.
A class may have additional constructors besides the primary constructor. These are defined by constructor definitions of the form:
def this(ps1)...(psn) = e.

6.7 Traits

Syntax:

```
TemplateDef ::= ‘trait’ TraitDef
TraitDef ::= id [TypeParamClause] TraitTemplateOpt
TraitTemplateOpt ::= ‘extends’ TraitTemplate | [[‘extends’] TemplateBody]
```

A trait is a class that is meant to be added to some other class as a mixin. Unlike normal classes, traits cannot have constructor parameters. Furthermore, no constructor arguments are passed to the superclass of the trait. Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of traits.

6.8 Object Definitions

Syntax:
ObjectDef ::= id ClassTemplate
An object definition defines a single object of a new class. Its most general form is object m extends t. Here, m is the name of the object to be defined, and t is a template that defines the base classes, behavior and initial state of m.

7 Pattern Matching

Pattern matching is the second most widely used feature of Scala, after function values and closures. Scala provides great support for pattern matching, in processing the messages. A pattern match includes a sequence of alternatives, each starting with the keyword case. Each alternative includes a pattern and one or more expressions, which will be evaluated if the pattern matches. An arrow symbol =¿ separates the pattern from the expressions.
Some examples of patterns are:

- The pattern ex: IOException matches all instances of class IOException, binding variable ex to the instance.
- The pattern (x, _) matches pairs of values, binding x to the first component of the pair. The second component is matched with a wildcard pattern.

- The pattern x :: y :: xs matches lists of length ¿= 2, binding x to the list's first element, y to the list's second element, and xs to the remainder.
- The pattern 1 |2 |3 matches the integers between 1 and 3.

8 Top Level Definitions

8.1 Compilation Units

A sequence of packages, classes and objects definitions and import clauses together make a compilation unit. Package p1; ... Package p2; Stats We can also write this like the compilation unit consisting of the packaging. Package p1... Package p2 Stats ....

8.2 Packaging

A set of classes, objects and packages is defined by a special object known as package. The set of members in it is defined by packaging. Package p Ds All the definition is ds in injected as members in the package with the help of packaging. These are called top level definitions' it is defined as private it is only visible to other members of the package. Package name cannot be as same as a class or a module name. These may not be used as values.

8.3 Package Objects

Syntax Package Object: = 'package' 'object' Objected Package object pratik extends agrawalla add the members of the template agrawalla to the package pratik. Per package we can only create one package object. In the directory of package pratik we should place the definition with the name package. Scala. We should not define a member with the same name of object or classes defined under the package. With the future version of Scala, they are trying to remove this limitation.

8.4 Genericity

Java programmers are well aware of the problem of genericity in programs. With the help of genericity, we can write code parametrized by types.
For example, a programmer writing a program for linked list library faces problem to decide the data type of element. List is always meant to use in different contexts, so we cannot restrict it to a particular data type.
Scala make it possible by defining generic class.

9 Regular Expressions

Scala supports regular expressions through Regex class available in the scala.util.matching package

Scala inherits its regular expression syntax from Java, which in turn inherits most of the features of Perl.
Some examples are-
z End of entire string
re+ Matches 1 or more of the previous thing
a—b Matches either a or b.

10 Exception Handling

10.1 Creating exception

You can create an exception object and then you throw it with thethrowkeyword as follows. throw new IllegalArgumentException

10.2 Throwing Exception

We can catch any exception in single block and then perform pattern matching against it using case blocks.