# 19CSE401 - Compiler Design

**Language Chosen –** Scala

**Group - 7**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Name** | **Roll No** |
| 1 | Penugonda Tandava Sai Naga Koushik | CB.EN.U4CSE19449 |
| 2 | Ravella Abhinav | CB.EN.U4CSE19453 |
| 3 | Singadi Shanthan Reddy | CB.EN.U4CSE19459 |

**Summary:**

Scala is a Java-like programming language which unifies object-oriented and functional programming. It is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes. Scala also supports a general notion of pattern matching which can model the algebraic types used in many functional languages.

# Important Features of Scala:

* It’s a high-level programming language
* It has a concise, readable syntax
* It’s statically-typed (but feels dynamic)
* It has an expressive type system.
* It’s a functional programming (FP) language
* It’s an object-oriented programming (OOP) language
* It supports the fusion of FP and OOP
* Contextual abstractions provide a clear way to implement term inference
* It runs on the JVM (and in the browser)
* It interacts seamlessly with Java code
* It’s used for server-side applications (including micro services), big data applications, and can also be used in the browser with Scala.js

# Keywords:

|  |  |
| --- | --- |
| **Name** | **Description** |
| abstract | Marks a class or trait as being abstract and uninstantiable. |
| case | Defines a matching pattern in match expressions and partial functions. |
| catch | Catches an exception. An alternate syntax that predates the utility monadic collection. |
| class | Defines a new class. |
| def | Defines a new method. |
| do | Part of the do..while loop definition. |
| else | The second part of an if..else conditional expression. |
| final | Marks a class or trait as being non-extendable. |
| finally | Executes an expression following a try block. |
| for | Begins a for-loop. |
| if | The first part of an if..else conditional expression, or the main part of an if conditional statement. |

|  |  |
| --- | --- |
| implicit | Defines an implicit conversion or parameter. |
| import | Imports a package, class, or members of a class to the current namespace. |
| lazy | Defines a value as being lazy, only defined the first time it is accessed. |
| match | Begins a match expression. |
| new | Creates a new instance of a class. |
| null | A value that indicates the lack of an instance. Has the type Null. |
| object | Defines a new object. |
| override | Marks a value or method as replacing the member of the same name in a base type. |
| package | Defines the current package, an incremental package name, or a package object. |
| private | Marks a class member as being inaccessible outside the class definition. |

|  |  |
| --- | --- |
| protected d | Marks a class member as being inaccessible outside the class definition or its subclasses. |
| return | Explicitly states the return value for a method. By default, the last expression in a method is used as the return value. |
| sealed | Marks a class as only allowing subclasses within the current file. |
| super | Marks a class member reference as one in the base type, versus one overridden in the current class. |
| this | Marks a class member reference as one in the current class, versus a parameter with the same name. |
| throw | Raises an error condition that breaks the current flow of operation and only resumes if the error is *caught* elsewhere. |
| trait | Defines a new trait. |
| true | One of the two Boolean values. |
| try | Marks a range of code for catching an exception. An alternate syntax that predates the util. Try monadic collection. |
| type | Defines a new type alias. |

|  |  |
| --- | --- |
| val | Defines a new, immutable value. |
| var | Defines a new, mutable variable. |
| while | Part of the do..while loop definition. |
| with | Defines a base trait for a class. |
| yield | Yields the return value from a for-loop. |

**Tokens:**

To construct tokens, characters are distinguished according to the following classes (Unicode general category given in parentheses):

1. Whitespace characters. \u0020 | \u0009 | \u000D | \u000A.
2. Letters, which include lower case letters (Ll), upper case letters (Lu), title case letters (Lt), other letters (Lo), letter numerals (Nl) and the two characters \u0024 ‘$’ and \u005F ‘\_’.
3. Digits ‘0’ | … | ‘9’.
4. Parentheses ‘(’ | ‘)’ | ‘[’ | ‘]’ | ‘{’ | ‘}’.
5. Delimiter characters ‘`’ | ‘'’ | ‘"’ | ‘.’ | ‘;’ | ‘,’.
6. Operator characters. These consist of all printable ASCII characters (\u0020 - \u007E) that are in none of the sets above, mathematical symbols (Sm) and other symbols (So).

**Semantics:**

Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e, that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

# Scala Type Inference:

# Scala Type Inference makes it optional to specify the type of variable provided that type mismatch is handled. With type inference capabilities, we can spend less time having to write out things compiler already knows. The Scala compiler can often infer the type of an expression so we don’t have to declare it explicitly.

Let us first have a look at the syntax of how immutable variables are declared in scala.

* val variable\_name : Scala\_data\_type = value

# Implicit Conversions in Scala:

Implicit conversions in Scala are the set of methods that are apply when an object of wrong type is used. It allows the compiler to automatically convert of one type to another.

Implicit conversions are applied in two conditions:

* First, if an expression of type A and S does not match to the expected expression type B.
* Second, in a selection e.m of expression e of type A, if the selector m does not represent a member of A.

In the first condition, a conversion is searched which is appropriate to the expression and whose result type matches to B. In the second condition, a conversion is searched which appropriate to the expression and whose result carries a member named m.

# 

# Type Casting in Scala:

A Type casting is basically a conversion from one type to another. In Dynamic Programming Languages like Scala, it often becomes necessary to cast from type to another.Type Casting in Scala is done using the **asInstanceOf[]** method.

Applications of asInstanceof method

* This perspective is required in manifesting beans from an application context file.
* It is also used to cast numeric types.
* It can even be applied in complex codes like communicating with Java and sending it an array of Object instances.

**CFG:**

**Start Symbol [S]:**

* Start

**Non-Terminals [N]:**

| Type | Example | CFG |
| --- | --- | --- |
| object | object obj\_Name  {  def main(args: )  {  <expr>  }  } | tmplDef  : 'case'? 'class' classDef  | 'case'? 'object' objectDef  | 'trait' traitDef  ;  objectDef  : Id classTemplateOpt  ;  classTemplateOpt  : 'extends' classTemplate  | ('extends'? templateBody)?  ;  classTemplate  : earlyDefs? classParents templateBody?  ; |
| For loop | for()  {  expressions  } | expr1  : 'if' '(' expr ')' NL\* expr ('else' expr)?  | 'while' '(' expr ')' NL\* expr  | 'try' expr ('catch' expr)? ('finally' expr)?  | 'do' expr 'while' '(' expr ')'  | 'for' ('(' enumerators ')' | '{' enumerators '}') 'yield'? expr  | 'throw' expr  | 'return' expr?  | ((simpleExpr | simpleExpr1 '\_'?) '.')? Id '=' expr  | simpleExpr1 argumentExprs '=' expr  | postfixExpr ascription?  | postfixExpr 'match' '{' caseClauses '}'  ;  simpleExpr1  : literal  | stableId  | '\_'  | '(' exprs? ')'  | simpleExpr '.' Id  | simpleExpr1 '\_'? '.' Id  | simpleExpr typeArgs  | simpleExpr1 '\_'? typeArgs  | simpleExpr1 argumentExprs  ; |
| While loop | while(condition) {  statement(a)  statement(b)  } | expr1  : 'if' '(' expr ')' NL\* expr ('else' expr)?  | 'while' '(' expr ')' NL\* expr  | 'try' expr ('catch' expr)? ('finally' expr)?  | 'do' expr 'while' '(' expr ')'  | 'for' ('(' enumerators ')' | '{' enumerators '}') 'yield'? expr  | 'throw' expr  | 'return' expr?  | ((simpleExpr | simpleExpr1 '\_'?) '.')? Id '=' expr  | simpleExpr1 argumentExprs '=' expr  | postfixExpr ascription?  | postfixExpr 'match' '{' caseClauses '}'  expr  : (bindings | 'implicit'? Id | '\_') '=>' expr  | expr1  ; |
| literals | val VariableName : DataType = [Initial Value] | literal  : '-'? IntegerLiteral  | '-'? FloatingPointLiteral  | BooleanLiteral  | CharacterLiteral  | StringLiteral  | SymbolLiteral  | 'null'  ; |
| Object | object MainObject{  def main(args:Array[String]){  }  } | objectDef  : Id classTemplateOpt  ; |
| Functions | def functionName ([list of parameters]) : [return type] = {  function body  return [expr]  } | funDef  : funSig (':' type\_)? '=' expr  | funSig NL? '{' block '}'  | 'this' paramClause paramClauses ('=' constrExpr | NL? constrBlock)  ; |
| Arrays | var z:Array[String] = new Array[String](size) | simpleExpr1  : literal  | stableId  | '\_'  | '(' exprs? ')'  | simpleExpr '.' Id  | simpleExpr1 '\_'? '.' Id  | simpleExpr typeArgs  | simpleExpr1 '\_'? typeArgs  | simpleExpr1 argumentExprs  ;  stableId  : Id  | stableId '.' Id  | (Id '.')? ('this' | 'super' classQualifier? '.' Id)  ; |
| Declaration Defination | var IntArray = Array(11, 15, 12, 14, 13)  var i: Int = 0  var j: Int = 0  var t: Int = 0 | patDef  : pattern2 (',' pattern2)\* (':' type\_)? '=' expr  ;  pattern2  : Id ('@' pattern3)?  | pattern3  ; |

**Parser Code:**

grammar hello;

start:equation;

equation:

    exp {System.out.println("success.");};

    exp: (header | variable | variable\_initialization1) functions\* |functions| SemiColon;

    header: header header|include\_macro|define\_macro|pragma\_macro;

    include\_macro: Hash Include (IncludeMethon1 | IncludeMethon2);

    define\_macro: Hash Define (Identifier param | Identifier);

    pragma\_macro: Hash Pragma (StartUp|Exit) Identifier;

    functions: (function\_definition | function\_declaration);

    function\_declaration:function\_type\_declaration (Identifier | (Mult Identifier)) (LPAREN params\_declaration | LPAREN) RPAREN SemiColon;

    function\_call:function\_type\_declaration (Identifier | (Mult Identifier)) (LPAREN params | LPAREN) RPAREN SemiColon;

    function\_definition:function\_type\_declaration (Identifier | (Mult Identifier)) (LPAREN params\_declaration | LPAREN) RPAREN LBrace (code RBrace|RBrace);

    code: function\_call |variable\_initialization|*for* |*if*|(expression SemiColon) | SemiColon| code (code|Return expression  SemiColon);

    params\_declaration:params\_declaration Comma params\_declaration|param\_declaration;

    param\_declaration: data\_type\_declaration Identifier;

    params:params Comma params|param;

    param: (Identifier | IntValue | FloatValue | CharValue);

    variable\_initialization: variable|variable\_initialization1|variable\_initialization2;

    variable: data\_type\_declaration (Identifier (Comma Identifier)\*) SemiColon;

    variable\_initialization1: data\_type\_definition  (Identifier|Identifier Equals expression) (Comma (Identifier|Identifier Equals expression))\* SemiColon;

    variable\_initialization2: Identifier assignment expression SemiColon;

*for*:(for1|for2|for3)(SemiColon|(LBrace (code RBrace|RBrace)));

    for1:For LPAREN ((variable\_initialization1|variable\_initialization2)|SemiColon) ((conditional\_expression SemiColon)|SemiColon) (((Identifier assignment expression)|(Inc|Dec)Identifier|Identifier(Inc|Dec))|SemiColon) RPAREN;

    for2: For LPAREN data\_type\_definition Identifier Colon Identifier RPAREN;

    for3: For LPAREN data\_type\_definition BitwiseAnd Identifier Colon Identifier RPAREN;

*if*: IF LPAREN if\_conditional RPAREN (LBrace code |LBrace) (RBrace| RBrace *else*);

    if\_conditional: LPAREN if\_conditional RPAREN|conditional|True|False| (Inc|Dec)Identifier|Identifier(Inc|Dec);

*else*: Else LBrace (code RBrace|RBrace);

    conditional:((conditional\_expression) (LessThan|LessThanEqualTo|EqualTo|NotEqual|GreaterThan|GreaterThanEqualTo) (conditional\_expression))|True|False;

    assignment:Equals|PlusEquals|MinusEquals|MultiEquals|DivEqulas|ModEquals|BAndEquals|BOrEquals|BXorEquals|BLeftShiftEquals|BRightShiftEquals;

expression:

    mathematical\_expression ;

    mathematical\_expression: mathematical\_expression (Inc|Dec) multiplicative\_exp|multiplicative\_exp;

    multiplicative\_exp: multiplicative\_exp (Mult|Div) additive\_exp|additive\_exp;

    additive\_exp: additive\_exp(Plus|Minus) shift\_exp|shift\_exp;

    shift\_exp:shift\_exp (LShit|RShift) relational\_exp|relational\_exp;

    relational\_exp: relational\_exp (LessThanEqualTo|LessThan|GreaterThanEqualTo|GreaterThan) equality\_exp|equality\_exp;

    equality\_exp: equality\_exp (EqualTo|NotEqual) b\_and\_exp|b\_and\_exp;

    b\_and\_exp: b\_and\_exp BitwiseAnd b\_xor\_exp|b\_xor\_exp;

    b\_xor\_exp: b\_xor\_exp BitwiseXor b\_or\_exp|b\_or\_exp;

    b\_or\_exp: b\_or\_exp BitwiseOr logical\_and\_exp|logical\_and\_exp;

    logical\_and\_exp: logical\_and\_exp LogicalAnd logical\_or\_exp|logical\_or\_exp;

    logical\_or\_exp: logical\_or\_exp LogicalOr conditional\_exp|conditional\_exp;

    conditional\_exp:conditional\_exp Conditional assignment\_exp|assignment\_exp;

    assignment\_exp:assignment\_exp assignment end|end;

    end: LPAREN mathematical\_expression RPAREN | (IntValue | FloatValue | CharValue| Identifier);

conditional\_expression:

    mathematical\_expression ;

    conditional\_mathematical\_expression: conditional\_mathematical\_expression (Inc|Dec) conditional\_multiplicative\_exp|conditional\_multiplicative\_exp;

    conditional\_multiplicative\_exp: conditional\_multiplicative\_exp (Mult|Div) conditional\_additive\_exp|conditional\_additive\_exp;

    conditional\_additive\_exp: conditional\_additive\_exp(Plus|Minus) conditional\_shift\_exp|conditional\_shift\_exp;

    conditional\_shift\_exp:conditional\_shift\_exp (LShit|RShift) conditional\_b\_and\_exp|conditional\_b\_and\_exp;

    conditional\_b\_and\_exp: conditional\_b\_and\_exp BitwiseAnd conditional\_b\_xor\_exp|conditional\_b\_xor\_exp;

    conditional\_b\_xor\_exp: conditional\_b\_xor\_exp BitwiseXor conditional\_b\_or\_exp|conditional\_b\_or\_exp;

    conditional\_b\_or\_exp: conditional\_b\_or\_exp BitwiseOr conditional\_logical\_and\_exp|conditional\_logical\_and\_exp;

    conditional\_logical\_and\_exp: conditional\_logical\_and\_exp LogicalAnd conditional\_logical\_or\_exp|conditional\_logical\_or\_exp;

    conditional\_logical\_or\_exp: conditional\_logical\_or\_exp LogicalOr conditional\_conditional\_exp|conditional\_conditional\_exp;

    conditional\_conditional\_exp:conditional\_conditional\_exp Conditional conditional\_assignment\_exp|conditional\_assignment\_exp;

    conditional\_assignment\_exp:conditional\_assignment\_exp assignment conditional\_end|conditional\_end;

    conditional\_end: LPAREN conditional\_mathematical\_expression RPAREN | (IntValue | FloatValue | CharValue| Identifier);

data\_type\_declaration:

    data\_type\_declaration\_decl {System.out.println("datatype.");};

    data\_type\_declaration\_decl: (data\_type\_declaration\_prefixes data\_type\_declaration\_dtype|data\_type\_declaration\_dtype);

    data\_type\_declaration\_dtype: Auto | Char | Double | Float | Int | Short | Void | Long;

    data\_type\_declaration\_prefixes: data\_type\_declaration\_prefixes data\_type\_declaration\_prefix |data\_type\_declaration\_prefix;

    data\_type\_declaration\_prefix: Const| Extern | Long | Static | Signed | Unsigned | Volatile;

data\_type\_definition:

    data\_type\_definition\_decl {System.out.println("datatype.");};

    data\_type\_definition\_decl: (data\_type\_definition\_prefixes data\_type\_definition\_dtype|data\_type\_definition\_dtype);

    data\_type\_definition\_dtype: Auto | Char | Double | Float | Int | Short | Void | Long;

    data\_type\_definition\_prefixes: data\_type\_definition\_prefixes data\_type\_definition\_prefix |data\_type\_definition\_prefix;

    data\_type\_definition\_prefix: Const| Long | Static | Signed | Unsigned | Volatile;

function\_type\_declaration:

    fun\_type\_decl {System.out.println("datatype.");};

    fun\_type\_decl: (fun\_type\_decl\_prefixes fun\_type\_decl\_dtype|fun\_type\_decl\_dtype);

    fun\_type\_decl\_dtype: Auto | Char | Double | Float | Int | Short | Void | Long;

    fun\_type\_decl\_prefixes: fun\_type\_decl\_prefixes fun\_type\_decl\_prefix |fun\_type\_decl\_prefix;

    fun\_type\_decl\_prefix: Const| Extern | Inline | Long | Static | Signed | Unsigned | Volatile;

Hash: '#';

Colon: ':';

Include: 'include';

IncludeMethon1: '"'[a-zA-Z][a-z.A-Z0-9]\*'"';

IncludeMethon2: '<'[a-zA-Z][a-z.A-Z0-9]\*'>';

Auto: 'auto';

Bool: 'bool';

Char: 'char';

Const: 'const';

Double: 'double';

Extern: 'extern';

Else: 'else';

Float:'float';

For:'for';

IF: 'if';

Int: 'int';

Inline: 'inline';

Long: 'long';

Short: 'short';

Signed: 'signed';

Static: 'static';

Unsigned: 'unsigned';

Void: 'void';

Volatile: 'volatie';

SemiColon: ';';

Comma: ',';

Return: 'return';

Define: 'define';

Pragma: 'pragma';

StartUp: 'startup';

Exit: 'exit';

True: 'true';

False: 'false';

Identifier: [a-zA-Z][a-zA-Z0-9]\*;

LShit: '<<';

RShift: '>>';

Inc: '++';

Dec: '--';

EqualTo: '==';

NotEqual: '!=';

LogicalAnd: '&&';

LogicalOr: '||';

GreaterThanEqualTo: '>=';

LessThanEqualTo: '<=';

PlusEquals: '+=';

MinusEquals: '-=';

MultiEquals: '\*=';

DivEqulas: '/=';

ModEquals: '%=';

BAndEquals: '&=';

BOrEquals: '|=';

BXorEquals: '^=';

BLeftShiftEquals: '<<=';

BRightShiftEquals: '>>=';

GreaterThan: '>';

LessThan: '<';

Equals: '=';

Plus: '+';

Minus: '-';

Mult: '\*';

Div: '/';

Mod: '%';

BitwiseAnd: '&';

BitwiseXor: '^';

BitwiseOr: '|';

Conditional: '?';

LPAREN: '(' {System.out.println("LParen");};

RPAREN: ')' {System.out.println("RParen");};

LBrace: '{' {System.out.println("LCurly");};

RBrace: '}' {System.out.println("RCurly");};

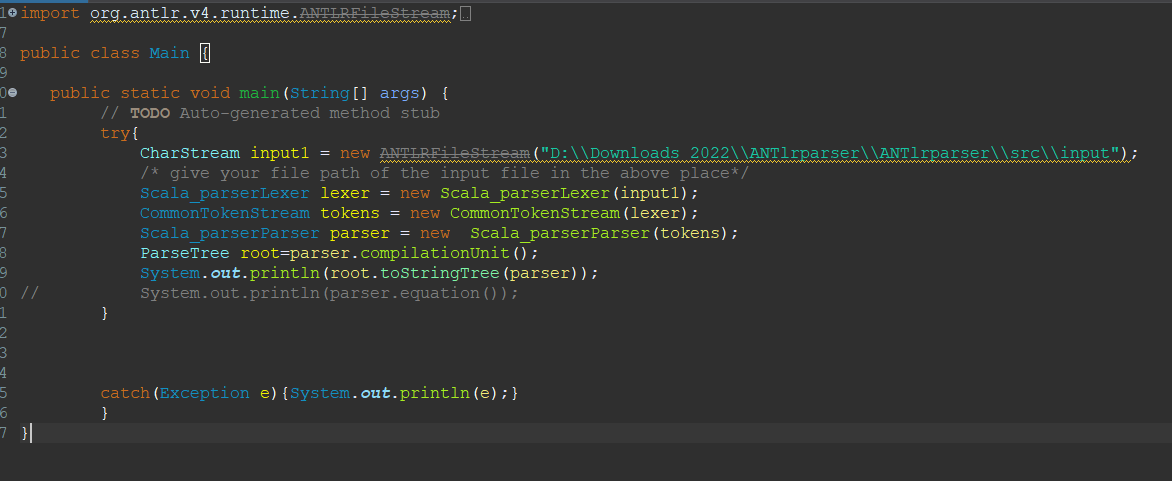
IntValue: ([0-9]+|'-'[0-9]+) {System.out.println("Number");};

CharValue: [a-zA-Z] {System.out.println("Char");};

FloatValue: ([0-9]+'.'[0-9]+|'-'[0-9]+'.'[0-9]+) {System.out.println("Float value");};

WS: [ \r\n\t] + -> skip;

**Main Program:**



**STEPS**

1. Generate ANTLR Recognizer
2. Run Java Application

**Example:**

1. **Input:**

object Palindrome {

*def* palindrome(x: String): String =

    {

        var rev: String = ""

*if* (x.toString.forall(\_.isDigit))

        {

*if*(x.toString.length > 1)

            {

                rev = x.toString.reverse

            }

        }

*else* *if*(x.length > 1)

        {

            rev = x.reverse

        }

*if* (rev == x) *return* ("Value is a palindrome")

*else* *return* ("Value is not a palindrome")

    }

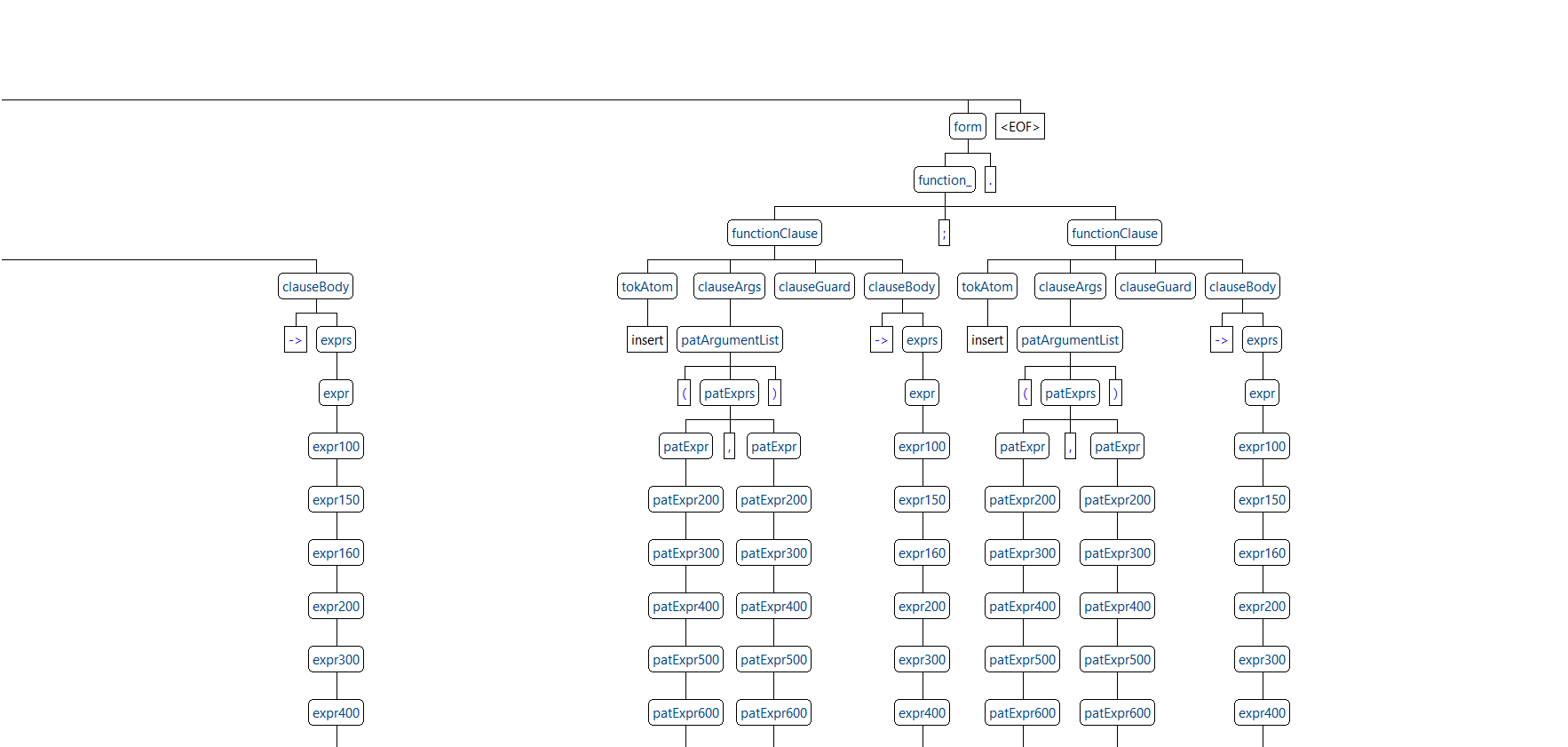
*def* main(args: Array[String]): Unit = {

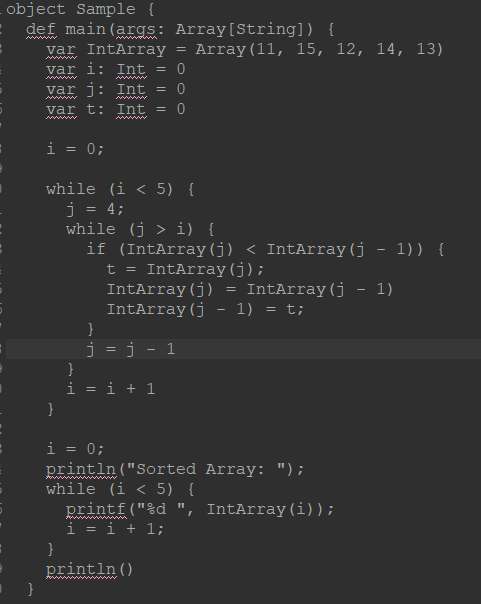
        println(palindrome(1234554321.toString))

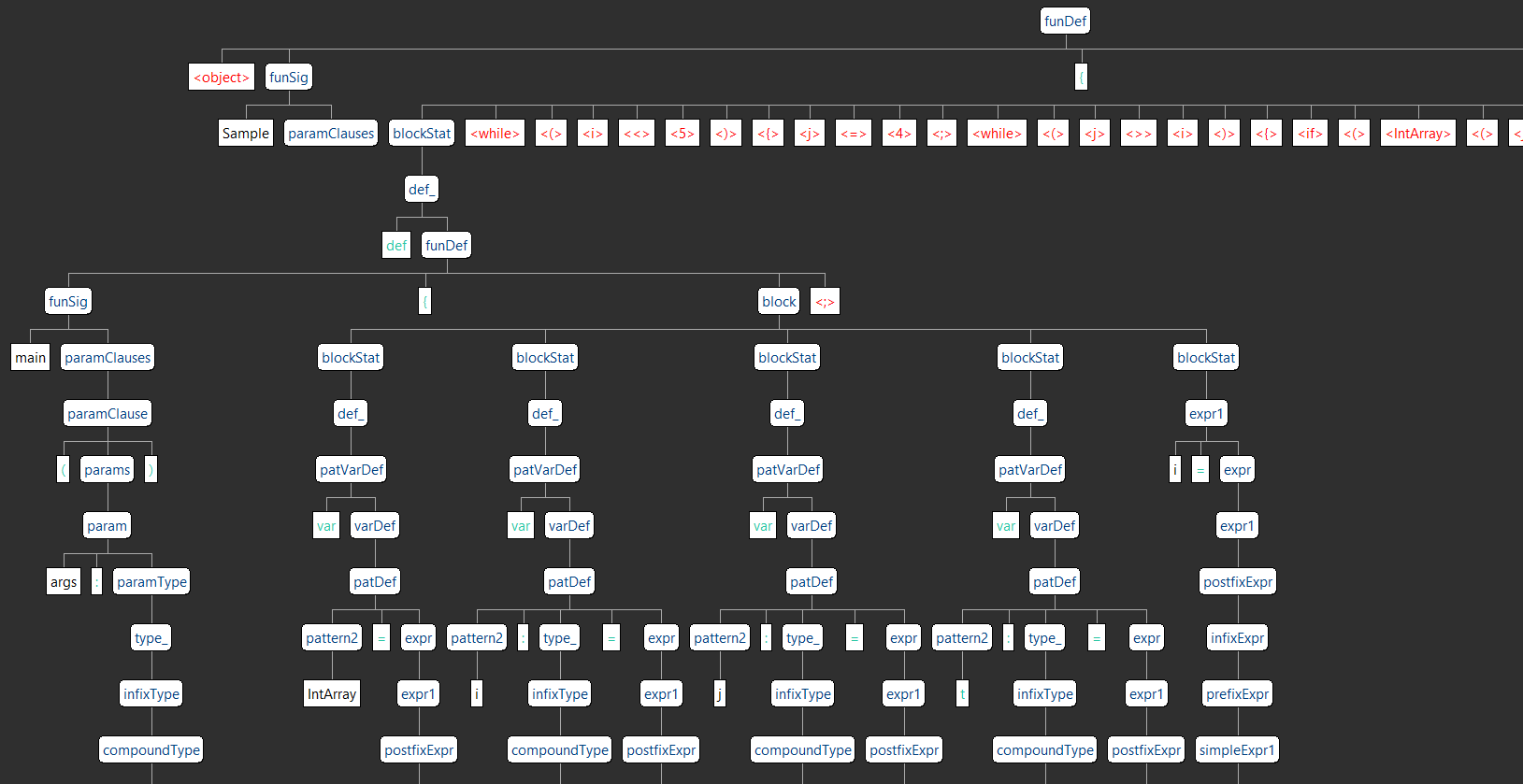
    }

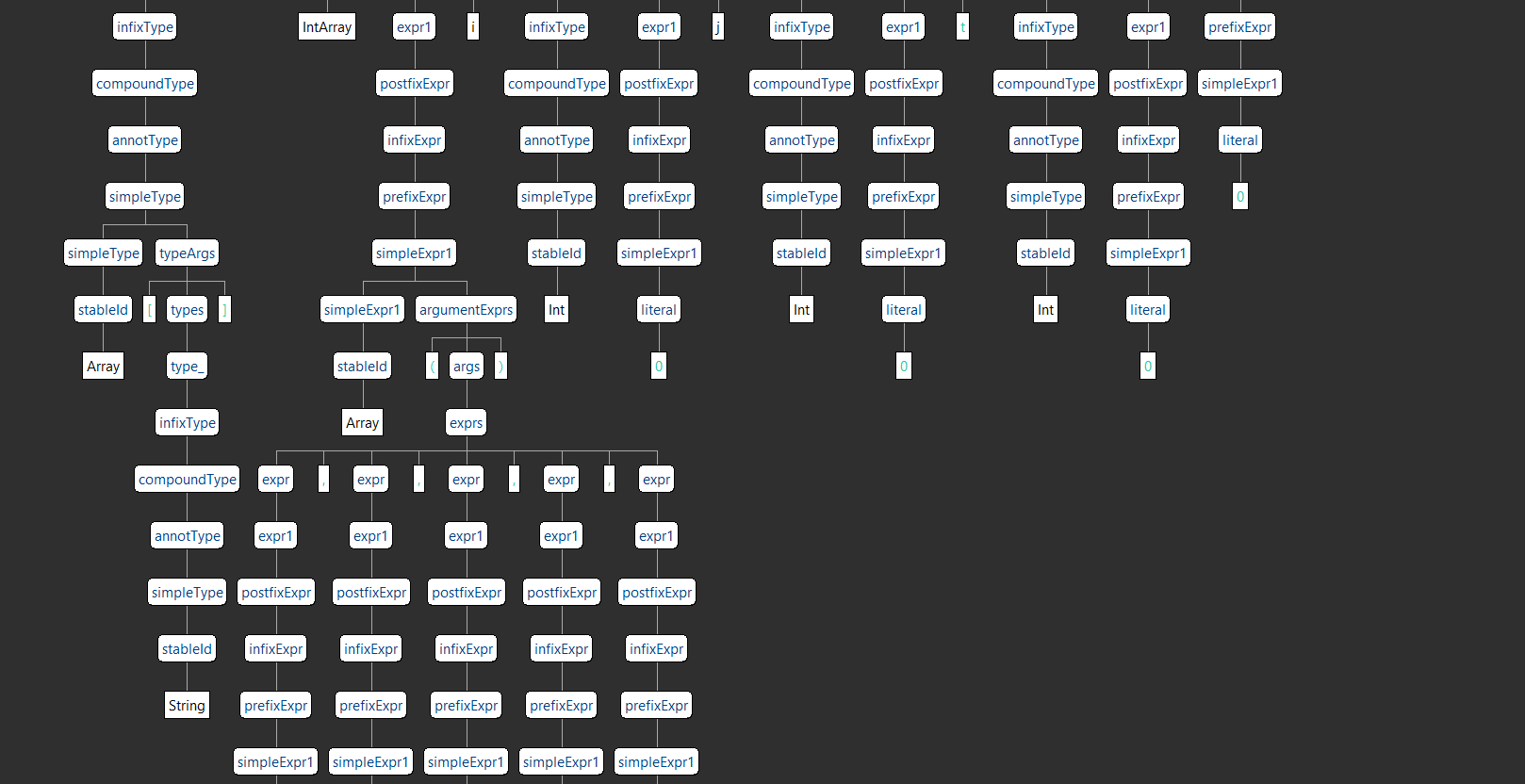
}

**Parse tree:**

****

****

**OUTPUT:**

****

**\*\*\***