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 dataapplications, 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.

protectedd	Marks a class member as being inaccessible outside the class definitionor 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 aparameter with the same name.
throw	Raises an error condition that breaks the current flow of operation andonly 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 syntaxthat 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 ;

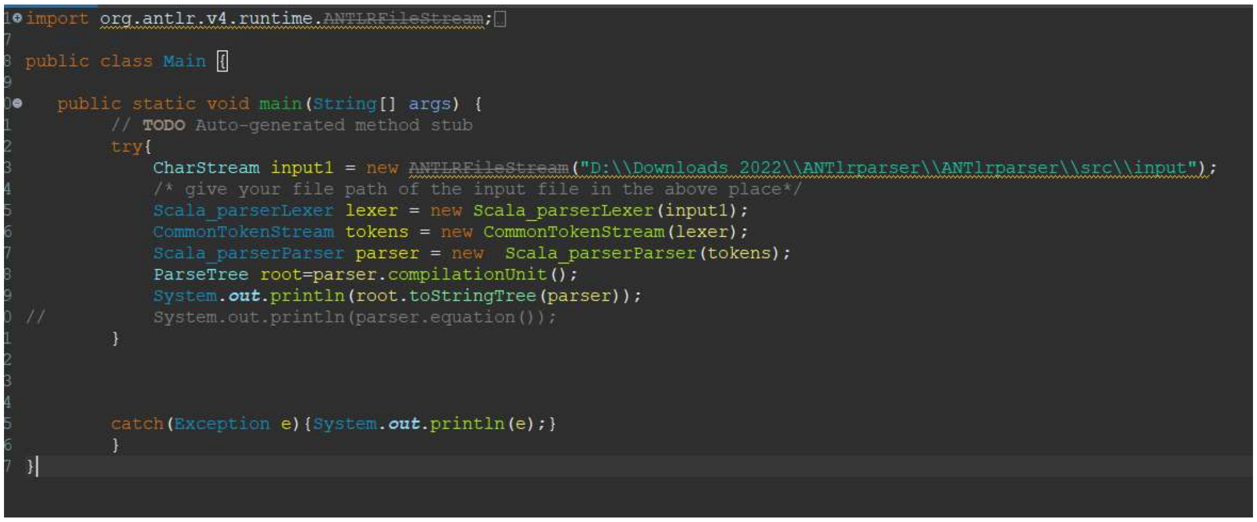
Parser Code:

```
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|BXorEqua
ls|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;
    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:



```
import org.antlr.v4.runtime.ANTLRFileStream;

public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        try{
            CharStream input1 = new ANTLRFileStream("D:\\Downloads 2022\\ANTlrparser\\ANTlrparser\\src\\input");
            /* give your file path of the input file in the above place*/
            Scala_parserLexer lexer = new Scala_parserLexer(input1);
            CommonTokenStream tokens = new CommonTokenStream(lexer);
            Scala_parserParser parser = new  Scala_parserParser(tokens);
            ParseTree root=parser.compilationUnit();
            System.out.println(root.toStringTree(parser));
//          System.out.println(parser.equation());
        }



        catch(Exception e){System.out.println(e);}
        }
}
```

STEPS

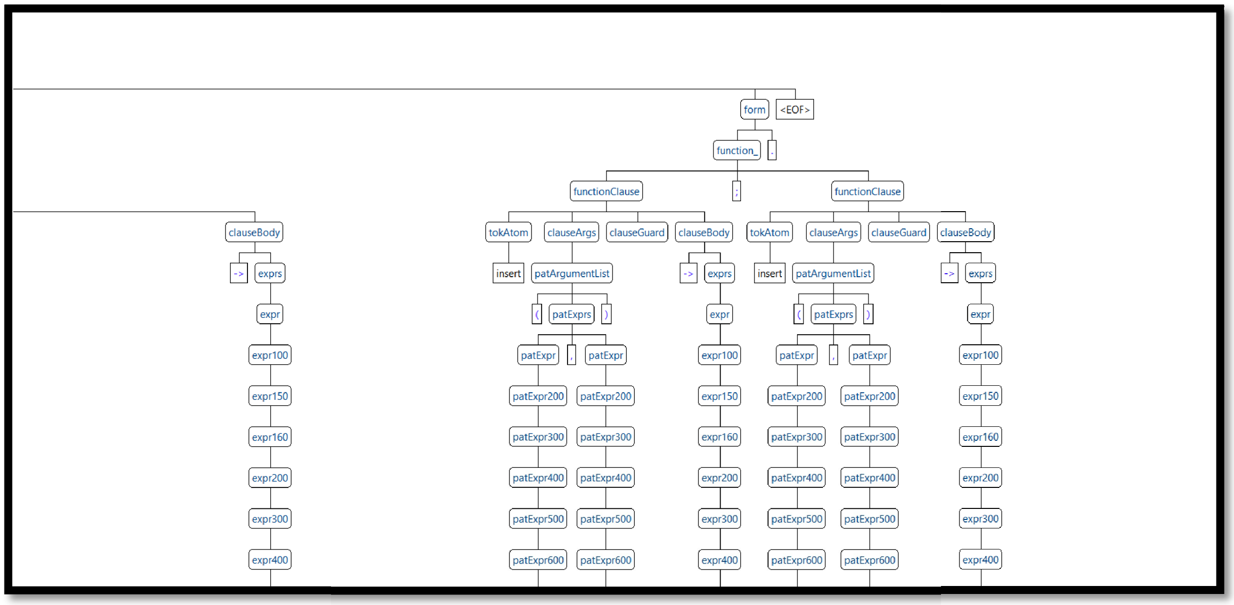
1. Generate ANTLR Recognizer
2. Run Java Application

Example:

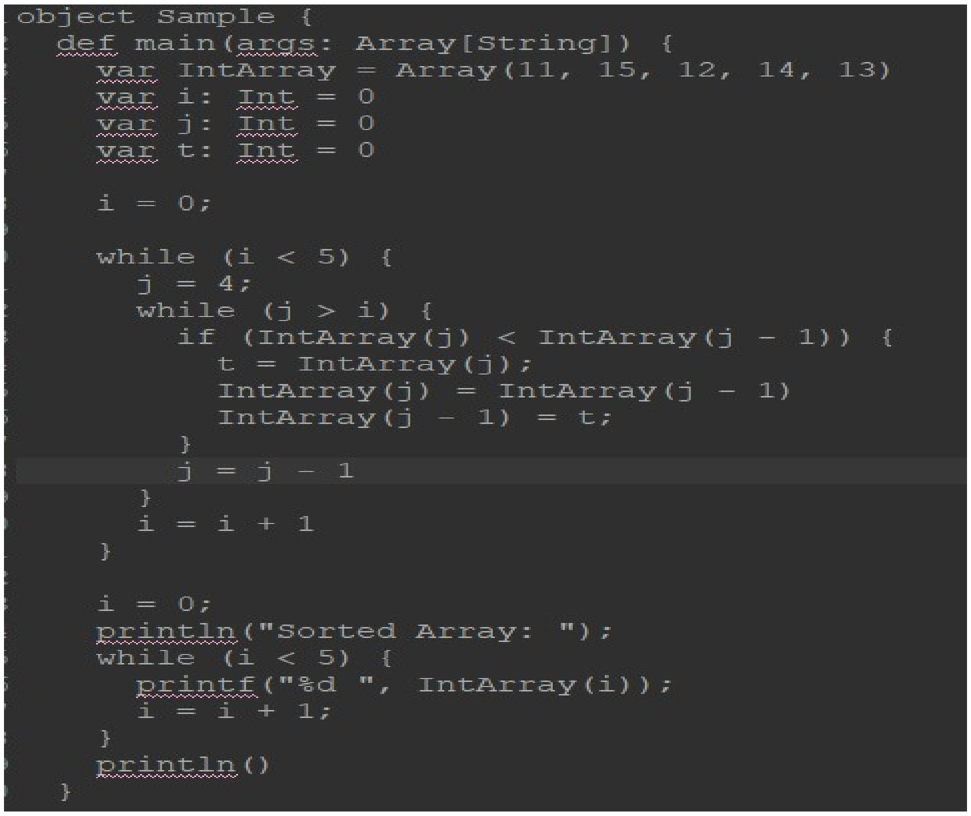
1. Input:

```
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:

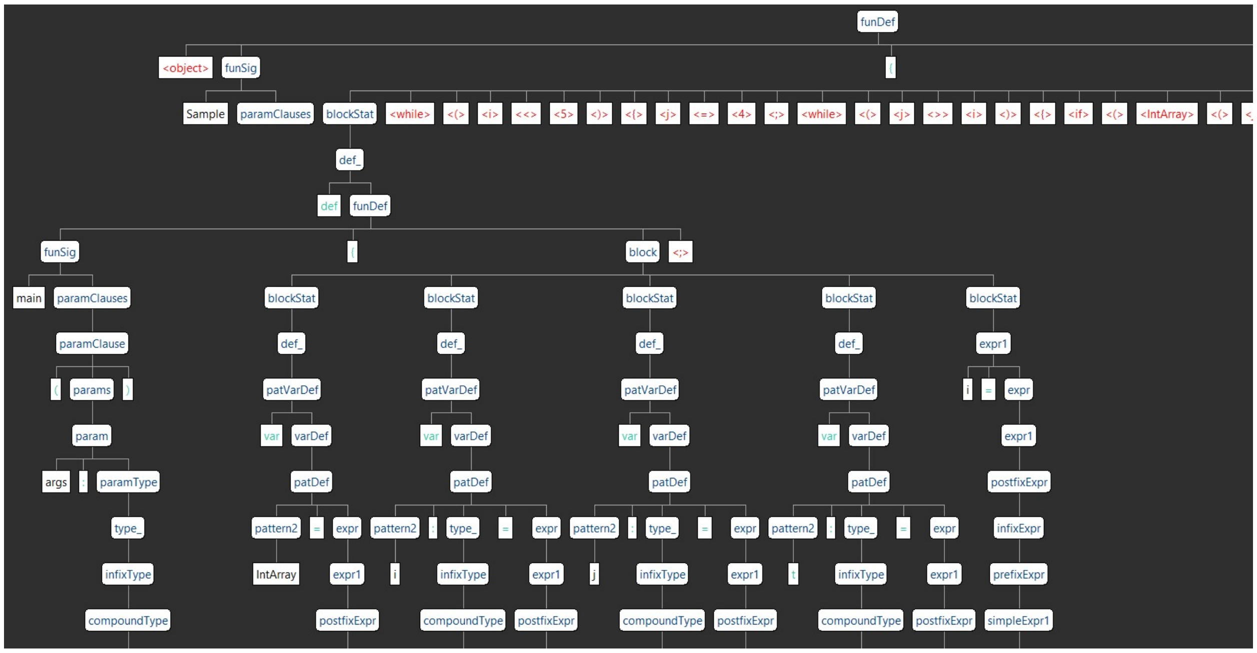


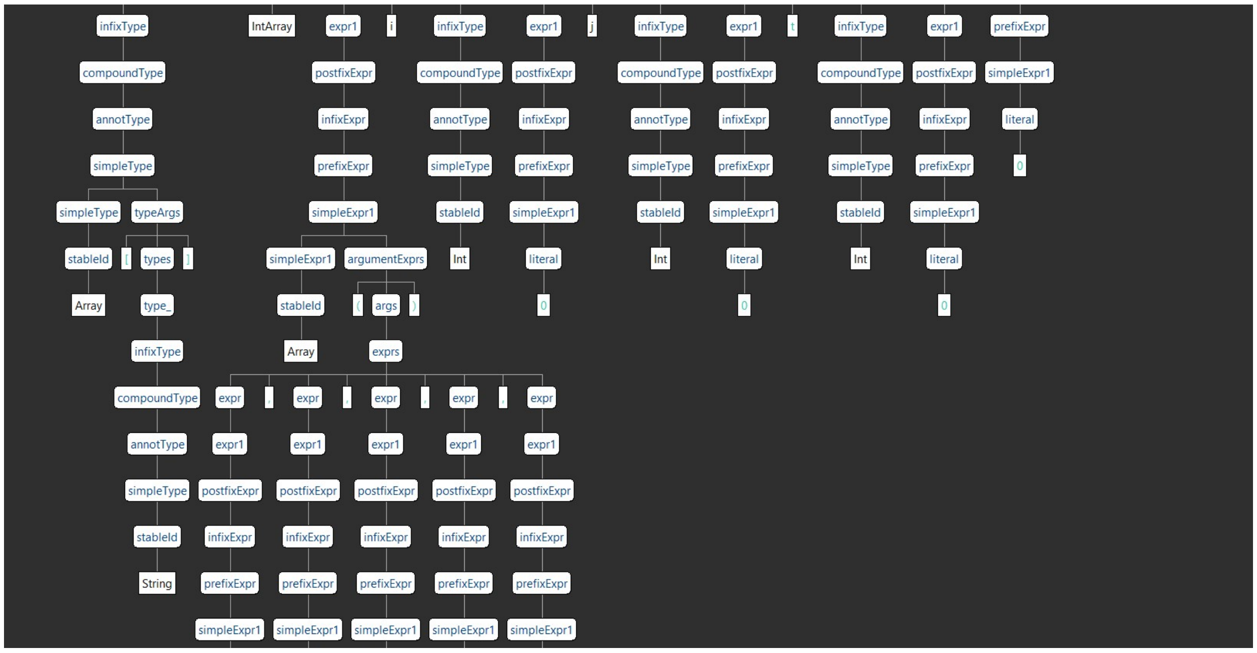
2.



```
object Sample {
  def main(args: Array[String]) {
    var IntArray = Array(11, 15, 12, 14, 13)
    var i: Int = 0
    var j: Int = 0
    var t: Int = 0

    i = 0;

    while (i < 5) {
      j = 4;
      while (j > i) {
        if (IntArray(j) < IntArray(j - 1)) {
          t = IntArray(j);
          IntArray(j) = IntArray(j - 1)
          IntArray(j - 1) = t;
        }
        j = j - 1
      }
      i = i + 1
    }

    i = 0;
    println("Sorted Array: ");
    while (i < 5) {
      printf("%d ", IntArray(i));
      i = i + 1;
    }
    println()
  }
```

OUTPUT:




infixType
compoundType
annotType
simpleType
simpleType
typeArgs
stableId
[
types
]
Array
type_
infixType
compoundType
annotType
simpleType
stableId
String
IntArray
expr1
postfixExpr
infixExpr
prefixExpr
simpleExpr1
simpleExpr1
argumentExprs
stableId
(
args
)
Array
exprs
expr
,
expr
,
expr
,
expr
,
expr
expr1
postfixExpr
infixExpr
prefixExpr
simpleExpr1
i
infixType
compoundType
annotType
simpleType
stableId
Int
expr1
postfixExpr
infixExpr
prefixExpr
simpleExpr1
literal
0
j
infixType
compoundType
annotType
simpleType
stableId
Int
expr1
postfixExpr
infixExpr
prefixExpr
simpleExpr1
literal
0
t
infixType
compoundType
annotType
simpleType
stableId
Int
expr1
postfixExpr
infixExpr
prefixExpr
simpleExpr1
literal
0
prefixExpr
simpleExpr1
literal
0

***