CONTEXT FREE GRAMMER

(CFG’s)

Sufiyan Mohammad Salman	***B20103065***
Abu Ubaida	***B20103007***
Muhammad Ibrahim khan	***B20103041***
Muhammad Shehroz	***B20103052***
Faraz Javed	***B20103022***
Muhammad Mairaj	***B20103043***

UBIT (BSSE)

Compiler construction

TERMINAL: Enclosed in single quote or without angle bracket such as <structure>, <EOF> , <Main>, <Cst> etc

Non-Terminal: Enclose in angle bracket such as ID, [], (), Main etc

CFG OF <Structure> And <EOF>:

```
<structure> --> static class ID <Inherit> { <S_Cst>
            --> <Final'> class ID <Inherit> { <G_Cst>
            --> interface  { <Int_St> } <Structure>
<EOF> --> ~
```

Description:

<Structure>: Represents the declaration of a structure or class. It can include various elements such as modifiers (static, final), the identifier (ID) for the structure or class, inheritance (Inherit), and a body of code (<S_Cst> or <G_Cst>) enclosed within curly braces ({}).

<Final'>: Represents an optional modifier final that can be used before declaring a structure or class. It indicates that the structure or class cannot be further subclassed or extended.

<G_Cst>: Represents the grammar for a class declaration within the structure. It includes the definition of member variables, member functions, and other class-related elements.

<Int_St>: Represents the grammar for interface declarations within the structure. It defines a contract or blueprint for implementing certain methods or behaviors.

<EOF>: Represents the end-of-file marker in the grammar, indicating the end of the input or source code.

In simple words, these CFG rules define the syntax for defining structures, classes, and interfaces in a programming language. They allow programmers to encapsulate related data and functionality within a structured format, facilitating better organization and modularity in code. The rules specify the usage of keywords like static, final, and interface to define different types of structures and classes. The body of a structure or class can include various elements such as variables, functions, and inheritance specifications. These rules are used in programming languages to support object-oriented programming paradigms and enable the creation of reusable and structured code components.

CFG OF <S_CST>:

```
<S_Cst> --> } <Structure>
        --> function <PubPriv'> static <Function_Sig> <S_Cst>
        --> private static <Class_Vars> <S_Cst>
        --> <Public'> static <S_Cst'>

<S_Cst'> --> DT <S_CstDT>
 --> ID <S_CstID>
--> void Main <S_Main>
```

Description:

In simple words, the purpose of this CFG is to define the syntax and structure for declaring and defining various elements within a program, such as structures, functions, class variables, and the Main method. It allows for the specification of access modifiers, data types, and identifiers for these program elements.

CFG OF <S_CstDT>:

```
<S_CstDT> --> ID <Declare'> <S_Cst>
 --> [] <S_CstDT'>
 --> Main <S_Main>

<S_CstDT'> --> ID <G_Arr'> <S_Cst>
 --> Main <S_Main>

<S_CstID> --> ID <Object'> <S_Cst>
 --> [] <S_CstID'>
 --> Main <S_Main>
 --> () <Body_MST> <S_Cst>
```

<S_CstID'> --> ID <O_Arr'> <S_Cst>

--> Main <S_Main>

Description:

In simple words, these CFG rules are used to define the syntax for declaring variables, arrays, objects, and the Main method within a program. They specify the structure and relationships between identifiers, data types, and array declarations. By following these rules, a parser or compiler can recognize and handle the appropriate syntax and semantics of these program elements.

CFG OF <S_MAIN>:

<S_Main> --> (<PL>) <BodyMST> <S_CstNM>

Description:

In simple words, this CFG rule defines the expected structure of the main method in a program, including its parameter list and body. It ensures that the main method is properly defined with the correct syntax, enabling the program to start execution from this point.

CFG OF <S_CstNM>:

<S_CstNM> --> } <Main_Done>

--> function <PubPriv'> static <Function_Sig> <S_CstNM>

--> private static <Class_Vars> <S_CstNM>

--> <Public'> static <ConsVar> <S_CstNM>

Description:

This CFG rule allows for the declaration of various class-level elements such as functions, static class variables, and constant variables that appear after the main method in a program. It defines the syntax and structure for these declarations within the class, enabling developers to define additional functionality and data within their programs.

CFG OF <ConsVar>

<ConsVar> --> DT <N_Dec>

--> ID <ConsVar'>

```
<ConsVar'> --> ( <AL> ) <Body_MST>
           --> ID <Object'>
           --> [] <O_Arr'>
```

Description:

These CFG rules allow for the declaration of constant variables with different characteristics. They define the syntax for declaring constant variables with explicit or inferred data types, and provide options to declare constants as functions with parameters, as object references, or as arrays. These rules enable programmers to define and use constant values that remain unchanged throughout the program execution.

CFG OF <C_Cst>:

```
<C_Cst>     --> } <Structure>
            --> <protectpriv> <SC> <Class_Vars> <C_Cst>
            --> function <AM> <Types> <C_Cst>
            --> <Class_Vars> <C_Cst>
            --> <Public'> <C_Cst'>

<C_Cst'>    --> static <C_Cst''>
            --> final <Class_Vars> <C_Cst>

<C_Cst''>   --> DT <C_CstDT>
            --> ID <C_CstID>
            --> void Main <GC_Main>
```

Description:

The purpose of the CFG rule <C_Cst> is to define the syntax for a class declaration

The purpose of the CFG rule <C_Cst'> is to extend the definition of class declarations

The purpose of the CFG rule <C_Cst''> is to extend the definition of static class declarations.

In simple words, these CFG rules define the syntax for declaring classes and various class-related elements in a programming language. They allow programmers to define classes with protected or private variables, functions or methods, class-level variables, and public elements.

The rules also provide the ability to declare static classes and specify their characteristics, such as data types, identifiers, and the presence of the main method.

CFG OF <C_CstDT> AND <C_CstID>:

<C_CstDT> --> ID <Declare'> <C_Cst>

--> [] <C_CstDT

--> Main <GC_Main>

<C_CstDT'> --> ID <G_Arr'> <C_Cst>

--> Main <GC_Main>

<C_CstID> --> ID <Object'> <C_Cst>

--> [] <C_CstID'>

--> Main <GC_Main>

--> (<AL>) <Body_MST> <C_Cst>

<C_CstID'> --> ID <O_Arr'> <C_Cst>

--> Main <GC_Main>

Description:

The purpose of the CFG rule <C_CstDT> is to define the syntax for declaring a data type in the context of class declarations

The purpose of the CFG rule <C_CstDT'> is to extend the definition of data type declarations within the context of class declarations.

The purpose of the CFG rule <C_CstID> is to define the syntax for identifying variables or objects within the context of class declarations.

The purpose of the CFG rule <C_CstDT'> is to extend the definition of data type declarations within the context of class declarations.

In simple words, these CFG rules define the syntax for declaring and identifying variables or objects within class declarations. They allow programmers to specify the data type of variables, including arrays, and identify objects or variables within the class context. These rules provide the grammar to declare and reference data types and identifiers, which are essential components in programming languages for variable declaration and manipulation.

CFG OF <GC_MAIN> AND <GC_NST> AND <GC_ConsVar>:

```
<GC_Main>    --> ( <AL> ) <BodyMST> <GC_Nst>


<GC_Nst>     --> } <Main_Done>

             --> function <AM> <Types> <GC_Nst>

             --> <PresPriv> <SC> <Class_Vars> <GC_Nst>

             --> <Public'> <GC_ConsVar> <GC_Nst>


<GC_ConsVar> --> final <Class_Vars>

             --> <Static'> <ConsVar>
```

DESCRIPTION:

The purpose of the CFG rule <GC_Main> is to define the syntax for the main method declaration.

The purpose of the CFG rule <GC_Nst> is to define the syntax for nested class declarations within the main method.

The purpose of the CFG rule <GC_ConsVar> is to define the syntax for constructor or variable declarations within the nested class.

In simple words, these CFG rules define the syntax for declaring the main method, nested classes, and their respective components within a programming language. They are used to specify the structure and organization of code within a main method, including the inclusion of nested classes, functions, and variables. These rules allow programmers to define nested class declarations and their associated elements, enabling them to create more complex and organized code structures within the main method.

CFG OF <G_Cst> And <G_Cst’> and <G_Cst’’>:

```
<G_Cst>      --> } <Structure>

             --> <PrivPres> <SC> <Class_Vars> <G_Cst>
```

--> function <AM> <Types> <G_Cst>

--> <Public'> <G_Cst'>

<G_Cst'> --> final <Class_Vars> <G_Cst>

--> <Static'> <G_Cst''>

<G_Cst''> --> DT <G_StDT>

--> ID <G_StID>

--> void Main <GC_Main>

DESCRIPTION:

The purpose of the CFG rule <G_Cst> is to define the syntax for a general constructor declaration.

The purpose of the CFG rule <G_Cst'> is to define the syntax for additional statements or components within the general constructor.

The purpose of the CFG rule <G_Cst''> is to define the syntax for different types of statements or components within the constructor

In simple words, these CFG rules define the syntax for declaring and defining a general constructor within a programming language. They allow programmers to specify the structure and behavior of a constructor, including the access modifier, class variables, statements, and other components within the constructor. These rules are used to define the construction logic of an object and enable the initialization of class variables or the execution of certain statements when an object of the class is created.

CFG OF <G_CstDT>, <G_CstDT'>, <G_CstID> And <G_CstID'>:

<G_CstDT> --> ID <Declare'> <G_Cst>

--> [] <G_CstDT'>

--> Main <GC_Main>

<G_CstDT'> --> ID <G_Arr'> <G_Cst>

--> Main <GC_Main>

<G_CstID> --> ID <Object'> <G_Cst>

```
                --> [] <G_CstID'>
                --> Main <GC_Main>
                --> ( <AL> ) <Body_MST> <G_Cst>
<G_CstID'>      --> ID <O_Arr'> <G_Cst>
                --> Main <GC_Main>
```

DESCRIPTION:

The purpose of the CFG rule <G_CstDT> is to define the syntax for declaring variables within the constructor.

The purpose of the CFG rule <G_CstDT'> is to define the syntax for additional statements or components related to the declared variables within the constructor.

The purpose of the CFG rule <G_CstID> is to define the syntax for using or assigning identifiers within the constructor.

The purpose of the CFG rule <G_CstID'> is to define the syntax for additional statements or components related to the identifiers within the constructor.

In simple words, these CFG rules define the syntax for declaring and using variables within a constructor, including regular variables, arrays, and object-related expressions. They allow programmers to specify the initialization or assignment of variables and their subsequent usage within the constructor. These rules are used to define the logic and behavior of a constructor in a programming language.

CFG OF <Main_Done>:

```
<Main_Done>  --> $
             --> static class ID <Inherit> { <S_CstNM>
             --> <ConcCond'> class ID <Inherit> { <GC_Nst>
             -->interface { <Int_St> } <Main_Done>
```

DESCRIPTION:

The purpose of the <Main_Done> rule is to indicate the completion of a main block or class declaration.

'$': This alternative represents the end of the main block or class declaration. The dollar sign symbol $ denotes the completion or termination point.

static class ID <Inherit> { <S_CstNM>: This alternative represents the declaration of a static class with an identifier (ID) and possible inheritance specified by <Inherit>, followed by a set of statements defined by <S_CstNM>. This alternative is used when defining a static class.

<ConcCond'> class ID <Inherit> { <GC_Nst>: This alternative represents the declaration of a class with an identifier (ID), possible inheritance specified by <Inherit>, followed by a set of statements defined by <GC_Nst>. This alternative is used when defining a regular class.

interface { <Int_St> } <Main_Done>: This alternative represents the declaration of an interface, indicated by the keyword interface, followed by a set of statements defined by <Int_St>. This alternative is used when defining an interface.

In simple words, this CFG rule serves as a marker for completing the main block or class declaration in a programming language. It indicates the end of the block or introduces the syntax for declaring a static class, regular class, or interface. It is used to define the structure and behavior of classes, interfaces, and their associated statements in a programming language.

CFG OF <AssignOp>:

```
<AssignOp>    --> =
              --> CompAssign
```

DESCRIPTION:

The purpose of the <AssignOp> rule is to represent the assignment operator, which is typically denoted by the equals sign (=). It is used to assign a value to a variable or an expression in programming. The assignment operator is a fundamental component of most programming languages.

CFG OF <Ref>:

```
<Ref>         --> . ID <Ref>
              --> [ <Exp> ] <Ref>
              --> ( <PL> ) . ID <Ref>
              --> $
```

DESCRIPTION:

The purpose of the <Ref> rule is to represent the referencing and accessing of elements, such as variables, properties, or methods, within a program. It allows you to specify how to navigate through data structures or objects to retrieve or modify specific elements.

The <Ref> rule consists of several alternatives:

.ID <Ref>: This alternative represents accessing an identifier (ID) within an object or data structure using dot notation. For example, obj.property accesses the property named property within the obj object.

[<Exp>] <Ref>: This alternative represents accessing an element within an array or collection using square brackets. The expression <Exp> inside the brackets specifies the index or key of the element to be accessed. For example, arr[0] accesses the element at index 0 in the array arr.

(<PL>) . ID <Ref>: This alternative represents accessing a method within an object or data structure using function call notation. The <PL> represents the parameter list for the method. For example, obj.method(arg1, arg2) calls the method method with the specified arguments on the obj object.

$: This alternative represents the end of the reference expression. It signifies that the reference operation is complete.

CFG OF <SST>:

```
<SST>          --> <For_St>
                 --> <If_St>
                 --> <While_St>
                --> <Return_St>
                 --> <Continue>
                 --> <Break>
                 --> <Try_St>
                 --> inc_dec <SP'> ID <Ref> ;
                 --> throw <Throw'>
                 --> <SP> ID <Ref> <AssignOP> <Exp> ;
                 --> DT <N_Dec>
                 --> ID <SST_ID>
```

```
<SST_ID>        --> ID <Object'>
                --> <SST_ID'>
<SST_ID'>       --> . ID <SST_ID'>
                --> [ <Exp> ] <SST_1>
                --> ( <PL> ) <SST_2>
                --> inc_dec ;
                --> <AssignOp> <Exp> ;
<SST_1>         --> . ID <SST_ID'>
                --> inc_dec ;
                --> <AssignOp> <Exp> ;
                --> ;
```

DESCRIPTION:

The purpose of the <SST> rule is to define the syntax for various types of statements within a program. It encompasses a range of statement types that can be executed to perform specific actions or control the flow of the program.

The <SST> rule includes several alternatives:

<For_St>: Represents a for loop statement, which allows you to repeatedly execute a block of code based on a specified condition.

<If_St>: Represents an if statement, which conditionally executes a block of code based on a specified condition.

<While_St>: Represents a while loop statement, which repeatedly executes a block of code as long as a specified condition is true.

<Return_St>: Represents a return statement, which terminates the execution of a function and returns a value.

<Continue>: Represents a continue statement, which skips the current iteration of a loop and continues with the next iteration.

<Break>: Represents a break statement, which immediately terminates the execution of a loop or switch statement.

<Try_St>: Represents a try-catch-finally statement, which allows you to handle exceptions and control the flow of the program in case of errors.

inc_dec <SP'> ID <Ref> ;: Represents an increment or decrement statement, which increments or decrements the value of a variable.

throw <Throw'>: Represents a throw statement, which is used to explicitly throw an exception.

<SP> ID <Ref> <AssignOP> <Exp> ;: Represents an assignment statement, which assigns a value to a variable or object property.

DT <N_Dec>: Represents a declaration statement, which declares a variable with a specified data type.

ID <SST_ID>: Represents an identifier followed by an optional object reference, indicating a statement related to a specific identifier.

In simple words, this CFG rule defines the syntax for various types of statements in a programming language. It covers statements related to loops, conditional execution, function returns, exception handling, variable assignments, declarations, and more. These statements are essential building blocks for controlling program flow, manipulating data, and performing different actions. This rule is commonly used in programming languages to specify the behavior and actions of a program.

CFG OF <SST_2>:

```
<SST_2>          --> ;
                 --> . ID <SST_ID'>
<Break>          --> Break ;
<Continue>       --> Continue ;
<While_St>       --> While ( <Exp> ) <Body>
<Body>           --> ;
                 --> <BodyMST>
<Try_St>         --> try <BodyMST> <CatchFinally>
<CatchFinally>   --> <Finally>
                 --> <Catch> <Finally'>
<Finally>        --> finally <BodyMST>
<Finally'>       --> finally <BodyMST>
                 --> $
<Catch>          --> catch ( <Exception> ID ) <BodyMST> <Catch'>
<Catch'>         --> <Catch>
```

--> $

DESCRIPTION:

<SST_2>: Represents a statement that consists of a single semicolon (;), indicating an empty statement or a statement with no specific action.

<Break>: Represents a break statement, which is used to terminate the execution of a loop or switch statement and exit the current iteration.

<Continue>: Represents a continue statement, which is used to skip the remaining code within a loop iteration and proceed to the next iteration.

<While_St>: Represents a while loop statement, which repeatedly executes a block of code as long as a specified condition is true.

<Body>: Represents a body of a statement or a block, which can either be an empty statement (;) or a sequence of statements enclosed within curly braces ({}).

<Try_St>: Represents a try statement, which is used to define a block of code that may potentially throw exceptions.

<CatchFinally>: Represents the handling of exceptions in a try-catch-finally block. It can either include a <Finally> block or both <Catch> and <Finally> blocks.

<Finally>: Represents a finally block, which contains a block of code that is always executed, regardless of whether an exception is thrown or caught.

<Finally'>: Represents an optional alternative for <Finally>, allowing it to either be a standalone <Finally> block or be followed by a termination symbol ($).

<Catch>: Represents a catch block, which is used to catch and handle specific exceptions that may be thrown within the corresponding try block.

<Catch'>: Represents an optional alternative for <Catch>, allowing it to either be a standalone <Catch> block or be followed by a termination symbol ($).

In simple words, these CFG rules define the syntax for statements related to exception handling, loops, and control flow in a programming language. They enable programmers to handle exceptions, control the flow of execution using loops, and define blocks of code to be executed under specific conditions. These rules are commonly used in programming languages to ensure proper error handling, implement looping constructs, and provide control over program execution flow.

IF - ELSE CONDITION:

<If_St>	--> if (<Exp>) <BodyMST> <OElse>
<OElse>	--> else <OElse'>
	--> $
<OElse'>	--> if (<Exp>) <BodyMST> <OElse>
	--> <BodyMST>

DESCRIPTION:

<If_St> **:** starting non terminal for if

< OElse >: non terminal for else for last if

<OElse'> **:** if within an else

FOR LOOP:

<For_St>	--> For (<St1> <St2> ; <St3>) <Body>
<St1>	--> <Dec>
	--> <AssignSt>
	--> ;
<Dec>	--> DT ID <Declare'>
<AssignSt>	--> <SP'> ID <Ref> <AssignOP> <Exp> ;
<St2>	--> $
	--> <Cond>
<St3>	--> inc_dec <Var>
	--> ID <For_Opt>
	--> $
<Cond>	--> ID <Cond'>
	--> <Const> <Cond'>
<Cond'>	--> $
	--> ROR <Exp>
<Return>	--> return <Ret'> ;

<Ret'>	--> $
	--> <Exp>
<For_Opt>	--> . ID <For_Opt>
	--> [<Exp>] <For_Opt'>
	--> (<PL>) . ID <For_Opt>
	--> inc_dec <For_Opt''>
	--> <AssignOp> <Exp> <For_Opt''>
<For_Opt'>	--> . ID <For_Opt>
	--> inc_dec <For_Opt''>
	--> <AssignOp> <Exp> <For_Opt''>
<For_Opt''>	--> $
	--> , ID <For_Opt>

Description:

<For_St>: for statement start

<St1> :declaration of iteration variable (starting point of iteration)

<St2> : condition checking non terminal

<St3> : increment and decrement in iterations

<Cond>: condition within <St2>

<Return>: returning any constant or expression

<For_Opt> : multiple operation increment decrement ex :object key values,inc/dec in multiple arithmetic operation or referencing variable inside func/object/class etc to be incremented or decremented

INHERITANCE:

<Inherit>	--> <Extends> <Implements>
<Extends>	--> Extends ID
	--> $
<Implements>	--> Implements ID <Implements'>
	--> $
<Implements'>	--> , ID <Implements'>
	--> $

Description:

<inherit>: inheritance statement start NT.

<Extends>: extending super class at multi/single/hierarchical level.

<Implements>: for reuasabilty of existing methods/variables in class.

INTEGER STATEMENT, PUBLIC /PRIVATE, LIST AND ARRAYS:

<Int_St>	--> <Public'> <RT> ID (<AL>) { } ;
	--> $
<Public'>	--> Public
	--> $
<RT>	--> DT <RT'>
	--> ID <RT'>
	--> Void
<RT'>	--> [ ]
	--> $
<AL>	--> $
	--> <AL'>

```
<AL'>              --> ID <AL'''> ID <AL''>
                   --> DT <AL'''> ID <AL''>
<Al''>             --> $
                   --> , <AL'>
<AL'''>            --> $
                   --> [ ]
<PubPriv'>         --> Public
                   --> Private
                   --> $
<Function_Sig>     --> <RT> ID ( <AL> ) <Body_MST>
<Class_Vars>       --> DT <N_Dec>
                   --> ID <O_Dec>


<N_Dec>            --> ID <Declare'>
                   --> [ ] ID <G_Arr'>
<O_Dec>            --> ID <Object'>
                   --> [ ] ID <O_Arr'>
<Declare'>         --> ;
                   --> , ID <Declare'>
                   --> = <Init_List>
<G_Arr'>           --> ;
                   --> , ID <G_Arr'>
                   --> = <Init_GArr> <G_Arr'>
<Init_GArr>        --> ID
                   --> new DT [ <Init_GArr'>
<Init_GArr'>       --> <Exp> ]
                   --> ] { <Val_GArr> }
<Val_GArr>         --> <Const> <Val_GArr'>
                   --> $
```

```
<Val_GArr'>          --> $
                     --> , <Const> <Val_GArr'>
<Init_List'>         --> ;
                     --> , ID <Declare>
<Init_List>          --> <SP'> ID <List1>
                     --> <Const> <List2>
                     --> ( <Exp> ) <List2>
                     --> ! <F> <Init_List'>
<List1>              --> = <Init_List>
                     --> . ID <List1>
                     --> [ <Exp> ] <List3>
                     --> ( <PL> ) <List2>
                     --> inc_dec <List2>
                     --> <List2>
<List2>              --> <T'> <E'> <C'> <B'> <A> <Init_List'>


<List3>              --> = <Init_List>
                     --> . ID <List1>
                     --> inc_dec <List2>
                     --> <List2>
```

Description:

<Int_St> -> <Public'> <RT> ID (<AL>) { } ; : This rule represents an integer statement. It consists of a visibility specifier, a return type, an identifier (function name), an optional parameter list enclosed in parentheses, an empty block statement, and a semicolon at the end.

<Public'> -> Public: This rule indicates that the visibility specifier can be the keyword "Public".

<Public'> -> Private: This rule indicates that the visibility specifier can be the keyword "Private".

<Public'> -> $: This rule indicates that the visibility specifier is optional, and if absent, it is represented by an empty string.

<RT> -> DT <RT'>: This rule represents the return type of a function. It consists of a data type followed by the rest of the return type.

<RT> -> ID <RT'>: This rule represents the return type of a function. It consists of an identifier (user-defined data type) followed by the rest of the return type.

<RT> -> Void: This rule represents the return type of a void function, indicated by the keyword "Void".

<RT'> -> [ ]: This rule represents an empty array return type, indicated by square brackets.

<RT'> -> $: This rule indicates that the return type does not have any additional elements.

<AL> -> $: This rule indicates that the parameter list can be empty.

<AL> -> <AL'>: This rule represents a non-empty parameter list. It consists of a parameter followed by the rest of the parameter list.

<AL'> -> ID <AL'''> ID <AL''>: This rule represents a parameter in the parameter list. It consists of an identifier (parameter name) followed by the rest of the parameter list.

<AL'> -> DT <AL'''> ID <AL''>: This rule represents a parameter in the parameter list. It consists of a data type followed by the rest of the parameter list.

<AL''> -> $: This rule indicates that the parameter list does not have any more parameters.

<AL''> -> , <AL'>: This rule represents the separation of parameters in the parameter list. It consists of a comma followed by another parameter.

<AL'''> -> $: This rule indicates that the parameter is not an array type.

<AL'''> -> [ ]: This rule indicates that the parameter is an array type, indicated by square brackets.

<PubPriv'> -> Public: This rule indicates that the visibility specifier can be the keyword "Public".

<PubPriv'> -> Private: This rule indicates that the visibility specifier can be the keyword "Private".

<PubPriv'> -> $: This rule indicates that the visibility specifier is optional, and if absent, it is represented by an empty string.

<Function_Sig> -> <RT> ID (<AL>) <Body_MST>: This rule represents the signature of a function. It consists of the return type, an identifier (function name), an optional parameter list enclosed in parentheses, and the function body.

<Class_Vars> -> DT <N_Dec>: This rule represents the declaration of class variables. It consists of a data type followed by a variable declaration.

<Class_Vars> -> DT <N_Dec>: This rule represents the declaration of class variables. It consists of a data type followed by a variable declaration.

<Class_Vars> -> ID <O_Dec>: This rule represents the declaration of class variables. It consists of an identifier (user-defined data type) followed by an object declaration.

<N_Dec> -> ID <Declare'>: This rule represents a variable declaration without initialization. It consists of an identifier (variable name) followed by any additional declarations.

<O_Dec> -> ID <Object'>: This rule represents an object declaration. It consists of an identifier (object name) followed by any additional declarations.

<Declare'> -> ; : This rule indicates the end of a variable declaration without initialization, represented by a semicolon.

<Declare'> -> , ID <Declare'>: This rule represents the separation of variables in a declaration. It consists of a comma followed by another variable declaration.

<Declare'> -> = <Init_List>: This rule represents a variable declaration with initialization. It consists of an equals sign followed by an initialization list.

<G_Arr'> -> ; : This rule indicates the end of an array declaration without initialization, represented by a semicolon.

<G_Arr'> -> , ID <G_Arr'> : This rule represents the separation of arrays in a declaration. It consists of a comma followed by another array declaration.

<G_Arr'> -> = <Init_GArr> <G_Arr'>: This rule represents an array declaration with initialization. It consists of an equals sign, an initialization list for the array, and another array declaration.

<Init_GArr> -> ID : This rule represents the initialization of an array with a single element. It consists of an identifier (element value).

<Init_GArr'> -> <Exp>] : This rule represents the initialization of an array with multiple elements. It consists of an expression followed by a closing square bracket.

<Init_GArr'> ->] { <Val_GArr> } : This rule represents the initialization of an array with a set of values. It consists of a closing square bracket, an opening brace, the values of the array separated by commas, and a closing brace.

<Val_GArr> -> <Const> <Val_GArr'> : This rule represents the values of an array during initialization. It consists of a constant value followed by any additional values.

<Val_GArr'> -> $: This rule indicates the end of the array initialization values.

<Val_GArr'> -> , <Const> <Val_GArr'>: This rule represents the separation of values in an array initialization. It consists of a comma followed by another constant value.

<Init_List'> -> ; : This rule indicates the end of a variable initialization list, represented by a semicolon.

<Init_List'> -> , ID <Declare> : This rule represents the separation of variables in an initialization list. It consists of a comma followed by another variable declaration.

<Init_List> -> <SP'> ID <List1> : This rule represents the initialization of a variable with a single value. It consists of a specific value (e.g., a constant) assigned to the variable.

<Init_List> -> <Const> <List2> : This rule represents the initialization of a variable with a constant value.

<Init_List> -> (<Exp>) <List2> : This rule represents the initialization of a variable with the result of an expression.

<Init_List> -> ! <F> <Init_List'>: This rule represents the initialization of a variable with a function call or object method call.

<List1> -> = <Init_List>: This rule represents the assignment of a specific value (e.g., a constant) to a variable.

<List1> -> . ID <List1> : This rule represents the assignment of a value obtained by accessing a member of an object or class using dot notation.

<List1> -> [<Exp>] <List3>: This rule represents the assignment of a value obtained by accessing an array element using an index.

<List1> -> (<PL>) <List2>: This rule represents the assignment of a value obtained by invoking a function or method with arguments.

<List1> -> inc_dec <List2>: This rule represents the assignment of a value obtained by incrementing or decrementing a variable.

<List1> -> <List2>: This rule represents the assignment of a value obtained by evaluating an expression.

<List2> -> <T'> <E'> <C'> <B'> <A> <Init_List'>: This rule represents the evaluation of an expression and any additional operations or assignments.

<List3> -> = <Init_List>: This rule represents the assignment of a specific value (e.g., a constant) to an array element.

<List3> -> . ID <List1>: This rule represents the assignment of a value obtained by accessing a member of an object or class using dot notation.

<List3> -> inc_dec <List2>: This rule represents the assignment of a value obtained by incrementing or decrementing an array element.

<List3> -> <List2>: This rule represents the assignment of a value obtained by evaluating an expression.

MULTI OPERATORS:

<Exp>	--> <B> <A>
<A>	--> || <B> <A>
	--> $
<B>	--> <C> <B'>
<B'>	--> && <C> <B'>
	--> $
<C>	--> <E> <C'>
<C'>	--> ROR <E> <C'>
	--> $
<E>	--> <T> <E'>
<E'>	--> PM <T> <E'>
	--> $
<T>	--> <F> <T'>
<T'>	--> MDM <F> <T'>
	--> $
<F>	--> (<Exp>)
	--> <Const>
	--> ! <F>
	--> <SP'> ID <OptF>

Description:

<Exp> -> <B> <A>: This rule represents an expression. It consists of a logical OR expression <B> followed by an optional logical OR operation <A>.

<A> -> || <B> <A>: This rule represents a logical OR operation. It consists of the logical OR operator || followed by another logical OR expression <B> and an optional logical OR operation <A>.

<A> -> $: This rule indicates that the logical OR operation is optional, and if absent, it is represented by an empty string.

<B> -> <C> <B'>: This rule represents a logical AND expression. It consists of a relational expression <C> followed by an optional logical AND operation <B'>.

<B'> -> && <C> <B'>: This rule represents a logical AND operation. It consists of the logical AND operator && followed by another relational expression <C> and an optional logical AND operation <B'>.

<B'> -> $: This rule indicates that the logical AND operation is optional, and if absent, it is represented by an empty string.

<C> -> <E> <C'>: This rule represents a relational expression. It consists of an arithmetic expression <E> followed by an optional relational operator and another arithmetic expression <C'>.

<C'> -> ROR <E> <C'>: This rule represents a relational operation. It consists of a relational operator (e.g., <, >, ==, etc.) followed by another arithmetic expression <E> and an optional relational operation <C'>.

<C'> -> $: This rule indicates that the relational operation is optional, and if absent, it is represented by an empty string.

<E> -> <T> <E'>: This rule represents an arithmetic expression. It consists of a term <T> followed by an optional arithmetic operation <E'>.

<E'> -> PM <T> <E'>: This rule represents an arithmetic operation. It consists of a plus or minus operator (+ or -) followed by another term <T> and an optional arithmetic operation <E'>.

<E'> -> $: This rule indicates that the arithmetic operation is optional, and if absent, it is represented by an empty string.

<T> -> <F> <T'>: This rule represents a term. It consists of a factor <F> followed by an optional term operation <T'>.

<T'> -> MDM <F> <T'>: This rule represents a term operation. It consists of a multiplication, division, or modulus operator (*, /, %) followed by another factor <F> and an optional term operation <T'>.

<T'> -> $: This rule indicates that the term operation is optional, and if absent, it is represented by an empty string.

<F> -> (<Exp>): This rule represents an expression enclosed in parentheses. It consists of an opening parenthesis, an expression <Exp>, and a closing parenthesis.

<F> -> <Const>: This rule represents a constant value.

<F> -> ! <F>: This rule represents the logical NOT operation. It consists of the ! operator followed by another factor <F>.

STATIC PROPERTY, OPTIONAL FIELD, ACCESS MODIFIERS, VALUE ASSIGNMENT AND INITIALIZATION OF ARRAY:

<SP>	--> This .
	--> Parent .
<SP'>	--> This .
	--> Parent .
	--> $
<SC>	--> Static
	--> Final’
	--> $
<OptF>	--> . ID <OptF>
	--> [<Exp>] <OptF>
	--> (<PL>) <OptF2>
	--> inc_dec
	--> $
<OptF1>	--> inc_dec
	--> . ID <OptF>
<OptF2>	--> . ID <OptF>
	--> $
<BodyMST>	--> { <MST> }
<MST>	--> <SST> <MST>
	--> $
<Object'>	--> ;
	--> , ID <Object'>
	--> = <Init_Obj> <Object'>

```
<Init_Obj>          --> new ID ( <PL> )
                    --> ID
<O_Arr'>            --> ;
                    --> , ID <O_Arr'>
                    --> = <Init_OArr> <O_Arr'>
<Init_OArr'>        --> ID
                    --> new ID [ <Init_OArr'>
<Init_OArr'>        --> ] { <Val_OArr> }
                    --> <Exp> ]
<Val_OArr>          --> new ID <PL> ) <Val_OArr'>
                    --> $

<Val_OArr'>         --> , new ID ( <PL> ) <Val_OArr'>
                    --> $
<PL>                --> $
                    --> <Exp> <PL'>
<PL'>               --> , <Exp> <PL'>
                    --> $
<Function_Sig>      --> <RT> ID ( <AL> ) <BodyMST>
<AM>                --> Public
                    --> Private
                    --> Protected
                    --> $
<Types>             --> <SC> <Function_Sig>

<protecPriv>        --> Protected
                    --> Private
<ConcCond'>         --> Final’
                    --> $
```

<Const>	--> Int_Const
	--> Float_Const
	--> String_Const
	--> Char_Const
	--> Bool_Const

<Final'>	--> Final
	--> $

Description:

<SP> -> This . : This rule represents the keyword "This" followed by a dot operator.

<SP'> -> This . : This rule represents the keyword "This" followed by a dot operator.

<SP'> -> Parent . : This rule represents the keyword "Parent" followed by a dot operator.

<SP'> -> $: This rule indicates that the dot operator is optional, and if absent, it is represented by an empty string.

<SC> -> Static: This rule represents the keyword "Static".

<SC> -> Final' : This rule represents the optional keyword "Final'".

<SC> -> $: This rule indicates that the static and final modifiers are optional, and if absent, they are represented by an empty string.

<OptF> -> . ID <OptF> : This rule represents the member access operation, where an identifier is accessed using dot notation.

<OptF> -> [<Exp>] <OptF> : This rule represents the member access operation, where an identifier is accessed using index notation.

<OptF> -> (<PL>) <OptF2> : This rule represents the member access operation, where a method is invoked with arguments.

<OptF> -> inc_dec : This rule represents the increment or decrement operation applied to an identifier.

<OptF> -> $: This rule indicates that the member access operation is optional, and if absent, it is represented by an empty string.

<OptF1> -> inc_dec : This rule represents the increment or decrement operation applied to an identifier.

<OptF1> -> . ID <OptF> : This rule represents the member access operation, where an identifier is accessed using dot notation.

<OptF2> -> . ID <OptF> : This rule represents the member access operation, where an identifier is accessed using dot notation.

<OptF2> -> $: This rule indicates that the member access operation is optional, and if absent, it is represented by an empty string.

<BodyMST> -> { <MST> } : This rule represents a block of code enclosed in curly braces.(multi line statement)

<MST> -> <SST> <MST> : This rule represents a sequence of statements.

<MST> -> $: This rule indicates the end of the statement sequence.

<Object'> -> ; : This rule represents the termination of an object declaration.

<Object'> -> , ID <Object'> : This rule represents the declaration of multiple objects.

<Object'> -> = <Init_Obj> <Object'> : This rule represents the initialization of objects.

<Init_Obj> -> new ID (<PL>) : This rule represents the initialization of an object using the "new" keyword with constructor arguments.

<Init_Obj> -> ID : This rule represents the initialization of an object using an existing identifier.

<O_Arr'> -> ; : This rule represents the termination of an array declaration.

<O_Arr'> -> , ID <O_Arr'> : This rule represents the declaration of multiple arrays.

<O_Arr'> -> = <Init_OArr> <O_Arr'> : This rule represents the initialization of arrays.

<Init_OArr'> -> ID : This rule represents the initialization of an array with a specified size.

<Init_OArr'> -> new ID [<Init_OArr'> : This rule represents the initialization of a dynamic array with an optional size.

<Init_OArr'> ->] { <Val_OArr> } : This rule represents the initialization of an array with specified values.

<Init_OArr'> -> <Exp>] : This rule represents the initialization of an array with a single value.

<Val_OArr> -> new ID <PL>) <Val_OArr'> : This rule represents the value assignment for an array element.

<Val_OArr'> -> , new ID (<PL>) <Val_OArr'> : This rule represents the assignment of multiple values for array elements.

<PL> -> $: This rule represents an empty parameter list.

<PL> -> <Exp> <PL'>: This rule represents a non-empty parameter list.

<PL'> -> , <Exp> <PL'> : This rule represents a list of expressions separated by commas.

<PL'> -> $: This rule indicates the end of the parameter list.

<Function_Sig> -> <RT> ID (<AL>) <BodyMST> : This rule represents the signature of a function, including the return type, function name, parameter list, and body.

<AM> -> Public : This rule represents the "public" access modifier.

<AM> -> Private : This rule represents the "private" access modifier.

<AM> -> Protected : This rule represents the "protected" access modifier.

<AM> -> $: This rule indicates that the access modifier is optional, and if absent, it is represented by an empty string.

<Types> -> <SC> <Function_Sig> : This rule represents the declaration of functions with static and final modifiers.

<protecPriv> -> Protected : This rule represents the "protected" access modifier.

<protecPriv> -> Private : This rule represents the "private" access modifier.

<ConcCond'> -> Final' : This rule represents the optional keyword "Final'".

<ConcCond'> -> $: This rule indicates that the "Final'" modifier is optional, and if absent, it is represented by an empty string.

<Const> -> Int_Const : This rule represents an integer constant.

<Const> -> Float_Const : This rule represents a float constant.

<Const> -> String_Const : This rule represents a string constant.

<Const> -> Char_Const : This rule represents a character constant.

<Const> -> Bool_Const : This rule represents a boolean constant.

<Final'> -> Final : This rule represents the "final" modifier.

<Final'> -> $: This rule indicates that the "final" modifier is optional, and if absent, it is represented by an empty string.