

1. Select the option that shows syntactically correct Neverlang constructs (that is, written with correct syntax).

• Option 1

```
module Addition{
  reference syntax {
    add1: AddExpression <-- AddExpression "+" Term;
    add2: AddExpression <-- Term;
  }
  role(execute) .{
    add1[0].value = add1[1].value + add1[2].value;
    add2[0].value = add2[1].value;
  }.
}
```

• Option 2

```
module Addition{
  reference syntax {
    add1: AddExpression <-- AddExpression "+" Term;
    add2: AddExpression <-- Term;
  }
  role(execute) {
    add1[0] = add1[1] + add1[2];
    add2[0] = add2[1];
  }
}
```

• Option 3

```
module Addition{
  reference syntax {
    add1: AddExpression <-- AddExpression "+" Term;
    add2: AddExpression <-- Term;
  }
  role(execute) {
    $add1 .{
      $add1[0] = $add1[1] + $add1[2];
    }.
    $add2 .{
      $add2[0] = $add2[1];
    }.
  }
}
```

• Option 4

```
module Addition{
  reference syntax {
    add1: AddExpression <-- AddExpression "+" Term;
    add2: AddExpression <-- Term;
  }
  role(execute) {
    add1: .{
```

```

        $add1[0].value = $add1[1].value + $add1[2].value;
    }.
    add2: .{
        $add2[0].value = $add2[1].value;
    }.
}
}

```

- Option 5

```

module Addition{
    reference syntax {
        add1: AddExpression = AddExpression + Term;
        add2: AddExpression = Term;
    }
    role(execute) {
        add1: .{
            $add1[0].value = $add1[1].value + $add1[2].value;
        }.
        add2: .{
            $add2[0].value = $add2[1].value;
        }.
    }
}

```

1. **Select the option that shows one or more reasonable Neverlang constructs, that is, constructs that make sense.**

- Option 1

```

module Addition{
    role(execute) .{
        BinaryOperation = Operand + Operand;
        BinaryOperation = Operand;
    }.
}

```

- Option 2

```

module Operation{
    reference syntax {
        op1: BinaryOperation <-- Operand Operand;
        op2: BinaryOperation <-- Operand;
    }
    role(execute) {
        op1: .{
            $op1[0].value = $op1[1].value + $op1[2].value;
        }.
        op2: .{
            $op2[0].value = $op2[1].value;
        }.
    }
    role(execute) {

```

```

    op1: .{
        $op1[0].value = $op1[1].value - $op1[2].value;
    }.
    op2: .{
        $op2[0].value = $op2[1].value;
    }.
}
}

```

- Option 3

```

module Operation{
    reference syntax {
        Op: BinaryOperation ← Operand Operand;
    }
    role(typePromotion){
        if($Op[1].type == Integer.class &&
            $Op[2].type == Float.class)
            Op: .{
                $Op[1].value = new Float((Integer)$Op[1].value);
            }.
        if($Op[2].type == Integer.class &&
            $Op[1].type == Float.class)
            Op: .{
                $Op[2].value = new Float((Integer)$Op[2].value);
            }.
    }
}

```

- Option 4

```

module Division{
    reference syntax {
        Fraction ← Dividend "/" Divisor;
    }
    role(execute) {
        0 .{
            $0.value = $1.value / $2.value;
        }.
    }
    role(checking){
        0 .{
            eval $2
            if($2.value == 0)
                throw new Error("Divisor can't be zero");
        }.
    }
}

```

- Option 5

```
module Expression{
  reference syntax {
    add1: AddExpression <-- AddExpression "+" Term;
    add2: AddExpression <-- Term;
  }
  reference syntax {
    sub1: SubExpression <-- SubExpression "+" Term;
    sub2: SubExpression <-- Term;
  }
}
```

1. **Select from the options presented below, the one that shows a set of Neverlang constructs with the following functionality: constructs that handle addition and multiplication operations with reverse Polish notation.**

The Polish inverse notation shows all operands first and finally all operators (e.g. the operation "two plus three" is written "2 3 +").

- Option 1

```
module binaryOperation{
  reference syntax {
    BinaryOperation <-- Operand Operand;
    BinaryOperation <-- Operand;
  }
}

module additionSem{
  reference syntax from binaryOperation
  role(evaluation) {
    0 .{
      $0.value = $1.value $2.value +
    }.
    3 .{
      $3.value = $4.value
    }.
  }
}

module multiplicationSem{
  reference syntax from binaryOperation
  role(evaluation) {
    0 .{
      $0.value = $1.value $2.value *
    }.
    3 .{
      $3.value = $4.value
    }.
  }
}
```

- Option 2

```

module additionSem{
  reference syntax from binaryOperation
  role(evaluation){
    0 .{
      $0.value = $1.value + $2.value
    }.
    3 .{
      $3.value = $4.value
    }.
  }
}

module multiplicationSem{
  reference syntax from binaryOperation
  role(evaluation){
    0 .{
      $0.value = $1.value * $2.value
    }.
    3 .{
      $3.value = $4.value
    }.
  }
}

module Addition{
  reference syntax{
    AddExpression <-- AddExpression Term "+" ;
    AddExpression <-- Term;
  }
}

module Multiplication{
  reference syntax{
    MulExpression <-- MulExpression Factor "*";
    MulExpression <-- Factor;
  }
}

```

- Option 3

```

module binaryOperation{
  reference syntax {
    BinaryOperation <-- Operand Operand;
    BinaryOperation <-- Operand;
  }
  role(evaluation){
    0 .{
      $0.value = $1.value * $2.value
    }.
    3 .{
      $3.value = $4.value
    }.
  }
}

```

```

    }.
  }
  role(evaluation){
    0 .{
      $0.value = $1.value + $2.value
    }.
    3 .{
      $3.value = $4.value
    }.
  }
}
module Addition{
  reference syntax{
    AddExpression <-- AddExpression Term "+" ;
    AddExpression <-- Term;
  }
}
module Multiplication{
  reference syntax{
    MulExpression <-- MulExpression Factor "*";
    MulExpression <-- Factor;
  }
}

```

- Option 4

```

module binaryOperation{
  reference syntax {
    BinaryOperation <-- Operand Operand;
    BinaryOperation <-- Operand;
  }
  role(evaluation) {
    when %(
      BinaryOperation <-- Operand Operand "+";
    )%
    0 .{
      $0.value = $1.value + $2.value
    }.
    3 .{
      $3.value = $4.value
    }.
    when %(
      BinaryOperation <-- Operand Operand "*";
    )%
    0 .{
      $0.value = $1.value * $2.value
    }.
    3 .{
      $3.value = $4.value
    }.
  }
}

module Addition{
  reference syntax{
    AddExpression <-- AddExpression Term "+" ;
    AddExpression <-- Term;
  }
}

module Multiplication{
  reference syntax{
    MulExpression <-- MulExpression Factor "*";
    MulExpression <-- Factor;
  }
}

```

- Option 5

```

module addition{
  reference syntax{
    AddExpression <-- AddExpression Term "+" ;
    AddExpression <-- Term;
  }
  role(evaluation) {
    0 .{
      $0.value = $1.value + $2.value
    }
  }
}

```

```

    }.
    3 .{
        $3.value = $4.value
    }.
}}

module multiplication{
    reference syntax{
        MulExpression <-- MulExpression Factor "*";
        MulExpression <-- Factor;
    }
    role(evaluation){
        0 .{
            $0.value = $1.value * $2.value
        }.
        3 .{
            $3.value = $4.value
        }.
    }
}

```

1. Select from the options presented below, the one that provides the correct meaning for the following set of Neverlang constructs (that is, the option that describes the functionality of the Neverlang LogLang language).

```

module Task{
    reference syntax {
        Task <-- "task" "{" CmdList "}" ;
        CmdList <-- Cmd CmdList;
        CmdList <-- Cmd ;
    }
}

module Rename{
    reference syntax {
        rnm: Rename <-- "rename" "(" String "," String ")";
        Cmd <-- Rename ;
    }
    role(execution) {
        rnm: .{
            String old = $rnm[1].string;
            String new = $rnm[2].string;
            $$FileOp.move(old,new)
        }.
    }
}

module Remove{
    reference syntax {
        rmv: Remove <-- "remove" "(" String ")";
        Cmd <-- Remove ;
    }
}

```

```

    role(execution) {
        rmv: .{
            String file = $rmv[1].string;
            $$FileOp.remove(file)
        }.
    }
}

module Backup{
    reference syntax {
        bkp: Backup <-- "backup" "(" String "," String ")" ;
        Cmd <-- Backup ;
    }
    role(execution) {
        bck: .{
            String src = $bck[1].string;
            String dest = $bck[2].string;
            $$FileOp.backup(src,dest)
        }.
    }
}

module logLangTypes{
    reference syntax {
        String <-- /\\" [ ^\\" ] \"/
    }
    role(lessing) {
        0 .{
            String stringText = #0.text;
        }
    }
}

endemic slice FileOpEndemic {
    declare {
        FileOp : mydirectory.loglang.utils.FileOp;
    }
}

language LogLang{
    slices
        Task
        Rename
        Remove
        Backup
        LogLangTypes
    roles syntax < lessing:execution
}

```

1. Option 1: The presented constructs define a language that performs file operations: rename files, remove files, and back up.
2. Option 2: constructs define and execute a set of tasks for which only abstract syntax is present.
3. Option 3: The presented constructs define a language that performs operations on strings.

4. Option 4: The presented constructs define a language that renames, removes, and rewrites strings in a file.
5. Option 5: constructs present syntax definition of a series of tasks without there being the complete syntax for each of them.

1. Below are examples of Neverlang constructs/keywords: endemic slices, the keyword "concrete syntax from" and the keyword "roles". You will be shown 5 options, each of which contains a statement about each of the constructs/keywords shown. You will need to identify the correct option, i.e. the option in which all three statements are true..

```
//ENDEMIC SLICE
endemic slice FileEndemic{
  declare {
    File : myFiles.FileEndemic;
  }
}
//CONCRETE SYNTAX FROM
slice addition{
  concrete syntax from additionSyntax
  module additionSemantics with role evaluation }
//ROLES
language InfixLanguage{
  slices
    InfixAddition
    InfixMultiplication
  roles syntax < :typePromotion:evaluation
}
```

1. Option 1:
  1. Endemic slices declare language types (F)
  2. concrete syntax from is a keyword that is used to add information to the syntax reference of a module (F)
  3. the keyword roles in language defines in what order the semantic roles will be executed (V)
2. Option 2:
  1. Endemic slices contain the declaration of variables or auxiliary methods that must be globally accessible from the code of each semantic action (V)
  2. The keyword "Concrete Syntax From" is used in forms to import concrete syntax from another module. (F)
  3. the keyword roles within language is used to identify semantic roles in reference to the slices present (F)
- Option 3:
  1. Endemic slices allow you to define globally accessible components to all slices. (V)

2. the keyword "concrete syntax from" is used within the slice to import the syntax reference from a module (V)
  3. The keyword roles in language defines which semantic roles are used and in what order they will be executed. The first role is always syntax. (V)
2. Option 4:
1. Endemic slices must always be defined in a language (F)
  2. the keyword "concrete syntax from" may not be present within a slice (F)
  3. roles are not always present within the language (F)
1. Option 5:
1. Endemic slices must have a reference syntax that defines their syntax (F)
  2. the keyword "concrete syntax from" is used in language to import the module that defines the construct syntax (f)
  3. The keyword roles in language is used to define the visit strategy of the syntactic tree (V)
1. **A number of Neverlang constructs are shown below. Identify among the options the set of constructs equivalent to those shown (i.e. those constructs that have the same functionality as those shown).**

```

module binaryOperation{
  reference syntax {
    op: BinaryOperation <-- Operand Operand;
    op1: BinaryOperation <-- Operand;
  }
}
module AdditionExpression {
  reference syntax from binaryOperation
  role(evaluation) {
    op: .{
      $op[0].value = $op[1].value + $op[2].value
    }.
    op1: .{
      $op1[0].value = $op1[1].value
    }.
  }
}
module MultiplicationExpression {
  reference syntax from binaryOperation
  role(evaluation) {
    op: .{
      $op[0].value = $op[1].value * $op[2].value
    }.
    op1: .{
      $op1[0].value = $op1[1].value
    }.
  }
}
module InfixAdditionSyntax{
  reference syntax{

```

```

    Op: AddExpression <-- AddExpression "+" Term ;
    Op1: AddExpression <-- Term;
  }
}
module InfixMultiplicationSyntax{
  reference syntax{
    Op: MulExpression <-- MulExpression "*" Factor ;
    Op1: MulExpression <-- Factor;
  }
}
slice infixAddition{
  concrete syntax from InfixAdditionSyntax
  module AdditionExpression with role evaluation
}
slice infixMultiplication{
  concrete syntax from InfixMultiplicationSyntax
  module MultiplicationExpression with role evaluation
}
language infixLang {
  slices
    infixAddition
    infixMultiplication
  roles syntax < evaluation
}

```

- Option 1

```

module AdditionExpression {
  reference syntax{
    AddExpression <-- AddExpression "+" Term;
    AddExpression <-- Term;
  }
  role(evaluation){
    0 .{
      $0.value = $1.value + $2.value
    }.
    3 .{
      $3.value = $4.value
    }.
  }
}
module MultiplicationExpression {
  reference syntax{
    MulExpression <-- MulExpression "*" Factor ;
    MulExpression <-- Factor;
  }
  role(evaluation){
    0 .{
      $0.value = $1.value * $2.value
    }.
    3 .{
      $3.value = $4.value
    }.
  }
}

```

```

}
language infixLang {
  slices
    AdditionExpression
    MultiplicationExpression
  roles syntax < evaluation
}

```

- Option 2

```

module binaryOperation{
  reference syntax {
    op: BinaryOperation <-- Operand Operand;
    op1: BinaryOperation <-- Operand;
  }
}
module AdditionExpression {
  reference syntax from binaryOperation
  role(evaluation){
    op: .{
      $op[0].value = $op[1].value + $op[2].value
    }.
    op1: .{
      $op1[0].value = $op1[1].value
    }.
  }
}
module MultiplicationExpression {
  reference syntax from binaryOperation
  role(evaluation){
    op: .{
      $op[0].value = $op[1].value * $op[2].value
    }.
    op1: .{
      $op1[0].value = $op1[1].value
    }.
  }
}
module AdditionSyntax{
  reference syntax{
    Op: AddExpression <-- AddExpression Term "+";
    Op1: AddExpression <-- Term;
  }
}
module MultiplicationSyntax{
  reference syntax{
    Op: MulExpression <-- MulExpression Factor "*";
    Op1: MulExpression <-- Factor;
  }
}
slice Addition{
  concrete syntax from AdditionSyntax
  module AdditionExpression with role evaluation
}

```

```

slice Multiplication{
  concrete syntax from MultiplicationSyntax
  module MultiplicationExpression with role evaluation
}
language infixLang {
  slices
    Addition
    Multiplication
  roles syntax < evaluation
}

```

- Option 3

```

module MultiplicationExpression {
  reference syntax{
    MulExpression <-- MulExpression "*" Factor ;
    MulExpression <-- Factor;
  }
  role(evaluation){
    0 .{
      $0.value = $1.value * $2.value
    }.
    3 .{
      $3.value = $4.value
    }.
  }
}
slice infixMultiplication{
  concrete syntax from MultiplicationExpression
  module MultiplicationExpression with role evaluation
}
language infixLang {
  slices
    infixMultiplication
  roles syntax < evaluation
}

```

- Option 4

```

module binaryOperation{
  reference syntax {
    op: BinaryOperation <-- Operand Operand;
    op1: BinaryOperation <-- Operand;
  }
}
module SubtractionExpression {
  reference syntax from binaryOperation
  role(evaluation){
    op: .{
      $op[0].value = $op[1].value - $op[2].value
    }.
    op1: .{
      $op1[0].value = $op1[1].value
    }.
  }
}

```

```

    }
}
module MultiplicationExpression {
  reference syntax from binaryOperation
  role(evaluation) {
    op: .{
      $op[0].value = $op[1].value * $op[2].value
    }.
    op1: .{
      $op1[0].value = $op1[1].value
    }.
  }
}
module InfixAdditionSyntax{
  reference syntax{
    Op: AddExpression <-- AddExpression "+" Term ;
    Op1: AddExpression <-- Term;
  }
}
module InfixMultiplicationSyntax{
  reference syntax{
    Op: MulExpression <-- MulExpression "*" Factor ;
    Op1: MulExpression <-- Factor;
  }
}
slice infixAddition{
  concrete syntax from InfixAdditionSyntax
  module AdditionExpression with role evaluation
}
slice infixMultiplication{
  concrete syntax from InfixMultiplicationSyntax
  module MultiplicationExpression with role evaluation
}
language infixLang {
  slices
    infixAddition
    infixMultiplication
  roles syntax < evaluation
}

```

- Option 5

```

module binaryOperation{
  reference syntax {
    op: BinaryOperation <-- Operand Operand;
    op1: BinaryOperation <-- Operand;
  }
}
module AdditionExpression {
  reference syntax from binaryOperation
  role(evaluation) {
    op: .{
      $op[0].value = $op[1].value + $op[2].value
    }.
  }
}

```

```

        op1: .{
            $op1[0].value = $op1[1].value
        }.
    }
}
module MultiplicationExpression {
    reference syntax from binaryOperation
    role(evaluation) {
        op: .{
            $op[0].value = $op[1].value * $op[2].value
        }.
        op1: .{
            $op1[0].value = $op1[1].value
        }.
    }
}
module InfixAdditionSyntax{
    reference syntax{
        Op: AddExpression <-- AddExpression "+" Term ;
        Op1: AddExpression <-- Term;
    }
}
module InfixMultiplicationSyntax{
    reference syntax{
        Op: MulExpression <-- MulExpression "*" Factor ;
        Op1: MulExpression <-- Factor;
    }
}

```

1. Select from the options shown below, the one that indicates the correct meaning for the module shown in the image below.

```

module Merge{
    reference syntax {
        mrg: Merge <-- "merge" "(" String "," String ")" ;
        Cmd <-- Merge ;
    }
    role(execution) {
        mrg: .{
            String file1 = $mrg[1].string;
            String file2 = $mrg[2].string;
            $$FileOp.merge(file1 ,file1)
        }.
    }
}

```

1. Option 1 The "Merge" module adds a semantic role to the language related to the Task

2. Option 2 The "Merge" module adds to the language the ability to rename a file with the merge command
  3. Option 3 The "Merge" module adds to the language the construct to create a new file from an old one
  4. Option 4 The "Merge" module allows you to add a command to the language, which merges two files into one.
  5. Option 5 The "Merge" module adds an abstract syntax for the merge construct to the language
1. **Select the option that provides the correct meaning for the following set of Neverlang constructs (that is, the option that describes the functionality of the following Neverlang language).**

```

language Desk {
  slices
    //Desk-specific language features
    Main
    Print
    Where
    //Terminal symbols
    Number
    Identifier
    //Assignments
    Assignment
    //Expressions
    AdditionSlice
    AddOperator
    Operand
  endemic slices
    //Variables
    SymbolTable
  roles syntax
    < terminal-evaluation //Initialize Numbers and
Identifiers by reading terminal symbols
    < populate           //Populate the symbol table
    < evaluation         //Evaluate and print the results
}
module Main {
  reference syntax {
    //Entry point: print <expression> where <assignments>
    Program <-- ExpressionsMain AssignmentsMain;
  }
  role(evaluation) {
    0 .{
      System.out.println("Declared Variables: " + $$Map);
      System.out.println("Result: " + $1.value);
    }.
  }
}
module Print {
  reference syntax {

```

```

        ExpressionsMain <-- "print" Expression;
        Expression <-- Addition;
    }
    //No need for roles, evaluation is performed on the
    Expression nonterminal
}
module Where {
    reference syntax {
        AssignmentsMain <-- "where" AssignemntList;
        //List of comma-separated assignments
        AssignemntList <-- Assignment "," AssignmentList;
        //Last element in the list of assigments
        AssignmentList <-- Assignment;
    }
    //No need for roles, the Map is populated when visiting the
    Assignment
}
module Number {
    reference syntax {
        //Integers only
        //$0      #0
        Number <-- /[0-9]+/;
    }
    role(terminal-evaluation) {
        0 .{
            //Translate the lexeme into an Integer value
            $0.value = Integer.parseInt(#0.text);
        }.
    }
}
module Identifier {
    reference syntax {
        //Lower-case and upper-case variable names
        //$0      #0
        Identifier <-- /[a-zA-Z]+/;
    }
    role(terminal-evaluation) {
        0 .{
            $0.id = #0.text;
        }.
    }
    role(evaluation) {
        0 .{
            //Retrieve variable value from symbol table
            $0.value = $$Map.get($0.id);
        }.
    }
}
module Assignment {
    reference syntax {
        //Assignments for the "where" clause: <id> = <number>
        Assignment <-- Identifier "=" Number;
    }
    role(populate) {
        0 .{

```

```

        //Fill the symbol table with the new variable
        $$Map.put((String) $1.id, (Integer) $2.value);
    }.
}
}
module AddOperator {
    reference syntax {
        op: AddOperator <-- "+";
    }
    role(evaluation) {
        op: .{
            //Addition is a function that takes two integer and
            returns another integer
            BiFunction<Integer,Integer,Integer> operator =
            (a,b) -> a+b;
            $op.operator = operator;
        }.
    }
}
module Addition {
    reference syntax {
        Addition <-- Addition AddOperator;
    }
}
slice AdditionSlice {
    //Combine the Addition syntax with the generic Binary
    Operation semantics
    concrete syntax from Addition
    module BinaryOp with role evaluation
}
module Operand {
    reference syntax {
        //Each element of the expression is either a number or a
        variable
        Factor <-- Number;
        Factor <-- Identifier;
    }
    //No roles are needed, the attributes are generated in the
    respective modules
}
module BinaryOp {
    reference syntax {
        op1: BinaryOperation <-- Operand Operator Operand;
        op2: BinaryOperation <-- Operand;
    }
    role(evaluation) {
        op1: .{
            //left operand
            Integer left = $op1[1].value;
            //right operand
            Integer right = $op1[3].value;
            //Retrieve the function performed by the infix
            operator
            BiFunction<Integer,Integer,Integer> operator =
            $op1[2].operator;

```

```

        //Measure and store the result
        $op1[0].value = operator.apply(left, right);
    }.
}
}
endemic slice SymbolTable {
    declare {
        Map: StrToIntMap;
    }
}

```

1. Option 1. The set of Neverlang constructs presented defines the Desk language, which is used to write a list of additions to files using the Print module.
2. Option 2. The set of Neverlang constructs presented defines the Desk language, which allows you to evaluate additions. In the Desk language, the global variable "Map" is defined using the endemic slice SymbolTable and is populated with a list of assignments when visiting the Assignment node.
3. Option 3. The set of Neverlang constructs presented evaluates binary expressions expressed with reverse Polish notation, as shown in the BynaryOp module.
4. Option 4 The set of Neverlang constructs presented defines a language that allows you to print on file a list of assignments "identifier = value", as shown by the Identifier module and the Number module.
5. Option 5 The set of Neverlang constructs presented defines a language that serves only to perform additions with 3 operators as shown in the role(evaluation) of the BinaryOp module.