

1. Selezionare l'opzione che mostra dei costrutti Neverlang sintatticamente corretti (ovvero, scritti con una sintassi corretta).

• Opzione 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;
  }.
}
```

• Opzione 2

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

• Opzione 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];
    }.
  }
}
```

• Opzione 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;
    }.
}
}

```

- Opzione 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;
        }.
    }
}

```

## 2. Selezionare l'opzione che mostra uno o più costrutti Neverlang ragionevoli, ovvero costrutti che hanno un senso.

- Opzione 1

```

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

```

- Opzione 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;
    }.
}
}

```

- Opzione 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);
            }.
    }
}

```

- Opzione 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");
        }.
    }
}

```

- Opzione 5

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

3. Selezioni tra le opzioni presentate di seguito, quella che mostra un insieme di costrutti Neverlang con la seguente funzionalità: costrutti che gestiscono le operazioni dell'addizione e moltiplicazione con notazione polacca inversa.

La notazione polacca inversa mostra prima tutti gli operandi e infine tutti gli operatori (per es. l'operazione "due più tre" viene scritta "2 3 +").

- Opzione 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
    }.
  }
}
```

- Opzione 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;
  }
}

```

- Opzione 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;
    }
}

```

- Opzione 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;
  }
}

```

- Opzione 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
    }.
  }
}

```

4. Selezioni tra le opzioni presentate di seguito, quella che fornisce il corretto significato per il seguente insieme di costrutti Neverlang (ovvero, l'opzione che descrive la funzionalità del linguaggio Neverlang LogLang).

```

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 = $bkp[1].string;
      String dest = $bkp[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
}
}

```

- Opzione 1: i costrutti presentati definiscono un linguaggio che esegue operazioni su file, ovvero rinomina file, rimuove file e fa il backup.
- Opzione 2: i costrutti presentano definizione ed esecuzione di una serie di compiti per cui però è presente solo una sintassi astratta.
- Opzione 3: i costrutti presentati definiscono un linguaggio che esegue operazioni su stringhe.
- Opzione 4: i costrutti presentati definiscono un linguaggio che rinomina, rimuove e riscrive delle stringhe in un file.
- Opzione 5: i costrutti presentano definizione della sintassi di una serie di compiti senza che ci sia la sintassi completa per ognuno di essi.

5. Di seguito sono mostrati degli esempi che riguardano dei costrutti/keyword di Neverlang: le endemic slice, la keyword “concrete syntax from” e la keyword “roles”. Le verranno mostrate 5 opzioni, ognuna delle quali contiene un'affermazione su ognuno dei costrutti/keyword mostrati. Lei dovrà identificare l'opzione corretta, ovvero l'opzione in cui tutte e tre le affermazioni sono vere.

```
//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
}
```

- Opzione 1:
  - Le endemic slices dichiarano i tipi del linguaggio (F)
  - concrete syntax from è una parola chiave che si usa per aggiungere informazioni alla reference syntax di un modulo (F)
  - la keyword roles in language definisce in quale ordine saranno eseguiti i ruoli semantici (V)
- Opzione 2:
  - Le endemic slices contengono la dichiarazione di variabili o dei metodi ausiliari che devono essere accessibili globalmente dal codice di ogni azione semantica (V)
  - la parola chiave “concrete syntax from” viene usata nei moduli per importare la sintassi concreta da un altro modulo. (F)
  - la keyword roles all'interno di language serve per identificare i ruoli semantici in riferimento alle slice presenti (F)
- Opzione 3:
  - Le endemic slices permettono di definire delle componenti accessibili globalmente a tutti gli slice. (V)
  - la parola chiave “concrete syntax from” è utilizzata all'interno della slice per importare la reference syntax da un modulo (V)
  - La keyword roles in language definisce quali sono i ruoli semantici usati e in quale ordine saranno eseguiti. Il primo ruolo è sempre syntax. (V)

- Opzione 4:
  - Le endemic slices devono essere sempre definite in un linguaggio (F)
  - la parola chiave “concrete syntax from” può non essere presente all’interno di una slice (F)
  - i roles non sono sempre presenti all’interno del language (F)
- Opzione 5:
  - Le endemic slices devono avere una reference syntax che ne definisce la sintassi (F)
  - La parola chiave “concrete syntax from” è utilizzata in language per importare il modulo che definisce la sintassi dei costrutti (F)
  - La keyword roles in language serve a definire la strategia di visita dell’albero sintattico (V)

**6. Di seguito sono mostrati una serie di costrutti Neverlang. Identifichi tra le opzioni l’insieme di costrutti equivalenti a quelli mostrati (ovvero quei costrutti che hanno le stesse funzionalità di quelli mostrati).**

```

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
}

```

- Opzione 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
}

```

- Opzione 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
  }

```

- Opzione 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
}

```

- Opzione 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
}

```

- Opzione 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;
  }
}

```

7. Selezioni tra le opzioni mostrate di seguito, quella che indica il corretto significato per il modulo mostrato nell'immagine di seguito.

```

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)
    }.
  }
}

```

- Opzione 1 Il modulo "Merge" aggiunge al linguaggio un ruolo semantico relativo al Task
- Opzione 2 Il modulo "Merge" aggiunge al linguaggio la possibilità di rinominare un file con il comando merge

- Opzione 3 Il modulo “Merge” aggiunge al linguaggio il costrutto per creare un nuovo file da uno vecchio
- Opzione 4 Il modulo “Merge” permette di aggiungere un comando al linguaggio, che fonde due file in uno solo.
- Opzione 5 Il modulo “Merge” aggiunge al linguaggio una sintassi astratta per il costrutto merge

**8. Selezionare l’opzione che fornisce il corretto significato per il seguente insieme di costrutti Neverlang (ovvero, l’opzione che descrive la funzionalità del seguente linguaggio Neverlang).**

```
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" AssignmentList;
    //List of comma-separated assignments
    AssignmentList <-- Assignment "," AssignmentList;
    //Last element in the list of assignments
    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;
  }
}

```

- Opzione 1. L'insieme di costrutti Neverlang presentati definisce il linguaggio Desk, che serve per scrivere su file una lista di addizioni tramite il modulo Print.
- Opzione 2. L'insieme di costrutti Neverlang presentati definisce il linguaggio Desk, che permette di valutare delle addizioni. Nel linguaggio Desk, la variabile globale "Map" è definita tramite l'endemic slice SymbolTable ed è popolata con una lista di assegnamenti quando si visita il nodo Assignment.
- Opzione 3. L'insieme di costrutti Neverlang presentati valuta delle espressioni binarie espresse con notazione polacca inversa, come mostra il modulo BynaryOp.
- Opzione 4 L'insieme di costrutti Neverlang presentati definisce un linguaggio che permette di stampare su file una lista di assegnamenti "identificatore = valore", come mostra il modulo Identifier e il modulo Number.
- Opzione 5 L'insieme di costrutti Neverlang presentati definisce un linguaggio che serve a effettuare solamente addizioni con 3 operatori come mostra il role(evaluation) del modulo BinaryOp.