Consider the example on lecture slides (slide 48, page 51)

We define a modulus, the computations are made in the field ZZ_modulus. We assume that all our intermediate computations will not overflow the "int" type in OCaml. Our computations also need the inverse of 2 in this field.

As the mod-function in OCaml returns negative answers when its first argument is negative, we will write a version that is usable for us.

```
[1]: let modulus = 17;;
    let inv2 = (modulus + 1) / 2;;

    let mmod x y =
        let r = x mod y
        in
        if r >= 0 then r else r + y;;
```

[1]: val modulus : int = 17

[1]: val inv2 : int = 9

[1]: val mmod : int -> int -> int = <fun>

The circuit is encoded by the following functions add_i, mult_i. Each tuple in the sets for add_i, mult_i gives rise to one summand of the form l1 * l2 * ... * ln, where li is either xi or (1-xi). For multiple elements, we can simplify by collecting similar factors. As the functions below work for any integer (considered modulo the modulus), they are already the multilinear extensions of add_i and mult_i.

As the number of arguments for different functions are different, we let all of them have the type [int]->int, where int is the type encoding also the field elements. For functions that have to work with a variable number of arguments, we give this number as an extra argument.

```
[2]: let add_1 = function [x1;x2;x3;x4;x5] -> mmod (x1 * x2 * (1-x3) * x4 * x5) \modulus | l -> raise (Failure ("add_1 was called with " ^ (string_of_int (List.length l)) ^ " arguments!"));;
```
In general, the multilinear extension of a function in a single point can be computed by evaluating it at all points in the hypercube.

Let the inputs to the circuit be the following:

```
let w3_raw = function [0;0] -> 6 | [0;1] -> 9 | [1;0] -> 12 | [1;1] -> 0 | l -> raise (Failure ("w3 was called with " ^ (string_of_int (List.length l)) ^ " arguments!"));
```

In general, the multilinear extension of a function in a single point can be computed by evaluating it at all points in the hypercube.
We need a mechanism for memoizing the function values. Otherwise all the following computations will refer back to the original add_i and mult_i and the computations of earlier multilinear extensions, and there will be a lot of recomputations.

We can now compute the values at other layers of the circuit. By using add_i, mult_i, the functions we define will immediately be multilinear.
let p00 = partial_compute (0 :: partial_b) (0 :: partial_c) and p01 = partial_compute (0 :: partial_b) (1 :: partial_c) and p10 = partial_compute (1 :: partial_b) (0 :: partial_c) and p11 = partial_compute (1 :: partial_b) (1 :: partial_c) in mmod (p00 + p01 + p10 + p11) modulus

let w2 = memo (compute_layer w3 add_3 mult_3 2);;
let w1 = memo (compute_layer w2 add_2 mult_2 2);;
let w0 = memo (compute_layer w1 add_1 mult_1 2);

[6]: val compute_layer :
  (int list -> int) ->
  (int list -> int) -> (int list -> int) -> int -> int list -> int = <fun>

[6]: val w2 : int list -> int = <fun>

[6]: val w1 : int list -> int = <fun>

[6]: val w0 : int list -> int = <fun>

The verifier knows the values in the input layer (w3) and in the output layer (w0). They are the following:

[7]: List.map (fun v -> (v, w3 v)) [[0;0];[0;1];[1;0];[1;1]]; List.map (fun v -> (v, w0 v)) [[0];[1]];;

[7]: - : (int list * int) list = 
  [[[0; 0], 6]; ([0; 1], 9); ([1; 0], 12); ([1; 1], 0)]

[7]: - : (int list * int) list = [[[0], 14]; ([1], 12)]

1 The beginning of the protocol

Verifier generates a random vector, the length of which is the length of addresses at layer 0. I.e.
the length of the vector is 1. The element(s) of this vector are randomly picked from the field, i.e.
y they are between 0 and (modulus-1).
Try to change the elements of the following vector.

```
[8]: let r0 = [7];;
```

```
[8]: val r0 : int list = [7]
```

The verifier also has a value $\text{ww0}$, which he believes to be equal to $(w0 \ r0)$. At the beginning of the protocol, he can compute $\text{ww0}$ himself.

```
[9]: let ww0 = w0 \ r0;;
```

```
[9]: val ww0 : int = 0
```

## 2 Checking layer no. 1

Prover and verifier run Sum-Check for the functions $W0$ and $W1$, shown on slide 54 (page 57) of the slides. In order to express the computations of this protocol in OCaml, we need to define some more helper functions. Below, $\text{sumF}$ will sum the function $f$ over all vectors with a given prefix $ll$ (with the elements in the suffix taking values either 0 or 1). The function $\text{interp}$ computes the coefficients of the quadratic function $a0 + a1 \cdot X + a2 \cdot X^2$, which is equal to $\text{sumF} f \ \text{numArgs} \ ll \ @ \ [X]$. It makes use of the helper function $\text{interpolate_parabola}$, which takes as its inputs the values of a quadratic function at points 0, 1, and (-1), and returns the coefficients of that parabola.

```
[10]: let rec sumF f numArgs = memo (function ll ->
      if (List.length ll) = numArgs then ll else mmod ((sumF f numArgs (ll \ @\ [0])) + (sumF f numArgs (ll \ @\ [1]))) modulus);
      
      let interpolate_parabola v0 v1 vm1 =
        let a2 = mmod ((v1 + vm1) * inv2 - v0) modulus
        in
        (v0, (mmod (v1 - a2 - v0) modulus), a2);
      
      let interp f numArgs ll =
        let at0 = sumF f numArgs (ll \ @\ [0])
        and at1 = sumF f numArgs (ll \ @\ [1])
        and atm1 = sumF f numArgs (ll \ @\ [-1])
        in
        interpolate_parabola at0 at1 atm1;
    
    [10]: val sumF : (int list -> int) -> int -> int list -> int = <fun>
    
    [10]: val interpolate_parabola : int -> int -> int -> int * int * int = <fun>
```
Verifier sends the random vector r0 to the Prover. The Prover has to convince him, that the summation in slide 54 (where the value of X is r0) is indeed equal to ww0.

At the beginning of Sum-Check, verifier initializes the value z0, which is the expected result of summation. This value is equal to ww0 we defined above.

```ocaml
let z0_1 = ww0;;
```

```
val z0_1 : int = 0
```

### 2.1 Round 1 of Sum-Checking on layer 1

The prover sends to the verifier the coefficients of the polynomial on the right-hand side in slide 54, where the variable X is at the position of the left bit of b.

```ocaml
let poly_1 = memo (function [x1;x2;x3;x4;x5] ->
  let w1b = w1 [x2;x3]
  and w1c = w1 [x4;x5]
  and a = add_1 [x1;x2;x3;x4;x5]
  and m = mult_1 [x1;x2;x3;x4;x5]
  in
  mmod (a * (w1b + w1c) + m * w1b * w1c) modulus
| _l -> raise (Failure ("poly_1 was called with " ^ (string_of_int (List.length_ ^ _l)) ^ " arguments!") ));;

let (a0_11, a1_11, a2_11) = interp poly_1 5 r0;;
```

```
val poly_1 : int list -> int = <fun>
```

```
val a0_11 : int = 1
val a1_11 : int = 16
val a2_11 : int = 16
```

Verifier checks whether $f(0) + f(1) = z0_1$, where $f(X) = a0_11 + a1_11 * X + a2_11 * X^2$.

```ocaml
z0_1 = mmod (a0_11 + (a0_11 + a1_11 + a2_11)) modulus;;
```

```
val - : bool = true
```

The verifier picks a random value r1 (try to change this value), and computes z1 as $f(r1)$.
Verifier sends the value of $r_1$ to the prover.

2.2 Round 2 of Sum-Checking on layer 1

The prover sends to the verifier the coefficients of the polynomial on the right-hand side in slide 54, where the variable $X$ is at the position of the right bit of $b$.

Verifier checks whether $f(0) + f(1) = z_1$, where $f(X) = a_{0_12} + a_{1_12} \cdot X + a_{2_12} \cdot X^2$.

The verifier picks a random value $r_2$ (try to change this value), and computes $z_2$ as $f(r_2)$.
2.3 Round 3 of Sum-Checking on layer 1

The prover sends to the verifier the coefficients of the polynomial on the right-hand side in slide 54, where the variable $X$ is at the position of the left bit of $c$.

\[ \text{let } (a_{0\_13}, a_{1\_13}, a_{2\_13}) = \text{interp poly}_1 \ 5 \ (r_0 @ [r_{1\_1}; r_{2\_1}]);; \]

\[ \text{val } a_{0\_13} : \text{int} = 16 \]
\[ \text{val } a_{1\_13} : \text{int} = 10 \]
\[ \text{val } a_{2\_13} : \text{int} = 7 \]

Verifier checks whether $f(0) + f(1) = z_{2\_1}$, where $f(X) = a_{0\_13} + a_{1\_13} \cdot X + a_{2\_13} \cdot X^2$.

\[ z_{2\_1} = \text{mmod } (a_{0\_13} + (a_{0\_13} + a_{1\_13} + a_{2\_13})) \text{ modulus};; \]

\[ - : \text{bool} = \text{true} \]

The verifier picks a random value $r_3$ (try to change this value), and computes $z_3$ as $f(r_3)$

\[ \text{let } r_{3\_1} = 9;; \]
\[ \text{let } z_{3\_1} = \text{mmod } (a_{0\_13} + a_{1\_13} \cdot r_{3\_1} + a_{2\_13} \cdot r_{3\_1} \cdot r_{3\_1}) \text{ modulus};; \]

\[ \text{val } r_{3\_1} : \text{int} = 9 \]
\[ \text{val } z_{3\_1} : \text{int} = 10 \]

Verifier sends the value of $r_3$ to the prover.

2.4 Round 4 of Sum-Checking on layer 1

The prover sends to the verifier the coefficients of the polynomial on the right-hand side in slide 54, where the variable $X$ is at the position of the right bit of $c$.

\[ \text{let } (a_{0\_14}, a_{1\_14}, a_{2\_14}) = \text{interp poly}_1 \ 5 \ (r_0 @ [r_{1\_1}; r_{2\_1}; r_{3\_1}]);; \]

\[ \text{val } a_{0\_14} : \text{int} = 0 \]
\[ \text{val } a_{1\_14} : \text{int} = 16 \]
\[ \text{val } a_{2\_14} : \text{int} = 11 \]

Verifier checks whether $f(0) + f(1) = z_{3\_1}$, where $f(X) = a_{0\_14} + a_{1\_14} \cdot X + a_{2\_14} \cdot X^2$.

\[ z_{3\_1} = \text{mmod } (a_{0\_14} + (a_{0\_14} + a_{1\_14} + a_{2\_14})) \text{ modulus};; \]
The verifier picks a random value $r_4$ (try to change this value), and computes $z_4$ as $f(r_4)$

```ocaml
let r4_1 = 3;;
let z4_1 = mmod (a_0_14 + a_1_14 * r4_1 + a_2_14 * r4_1 * r4_1) modulus;;
```

The Verifier would like to evaluate poly_1 at the position $(r_0 @ [r_1_1;r_2_1;r_3_1;r_4_1])$, but he cannot, because he does not know the values $w_1 [r_1_1;r_2_1]$ and $w_1 [r_3_1;r_4_1]$. Instead, the prover has to help.

### 2.5 Going from layer 1 to layer 2

Define the line $l_1$, going through the points $[r_1_1;r_2_1]$ and $[r_3_1;r_4_1]$. The prover tells the verifier the coefficients of the polynomial $q_1 = w_1 . l_1$, where the dot denotes composition. As $w_1$ takes two arguments, the degree of $q_1$ is 2.

```ocaml
let l1 x = [\text{mmod} (\text{r}_1 + (\text{r}_3_1-\text{r}_1) \times x) \modulus; \text{mmod} (\text{r}_2_1 + (\text{r}_4_1-\text{r}_2_1) \times x) \modulus];
let (q1_0, q1_1, q1_2) = \text{interpolate_parabola} (w1 (l1 0)) (w1 (l1 1)) (w1 (l1 (-1)));;
```

Verifier can now complete the Sum-Check, using $q_1(0)$ and $q_1(1)$ as the values $w_1 [r_1_1;r_2_1]$ and $w_1 [r_3_1;r_4_1]$.

```ocaml
let w1b = q1_0
and w1c = mmod (q1_0 + q1_1 + q1_2) modulus
and a = add_1 (r0 @ [r1_1;r2_1;r3_1;r4_1])
and m = mult_1 (r0 @ [r1_1;r2_1;r3_1;r4_1])
in z4_1 = mmod (a * (w1b + w1c) + m * w1b * w1c) modulus;;
```
Verifier picks a random $r\#1$ (try to change this value), defines $r_1 = l_1(r\#1)$ and $ww_1 = q_1(r\#1)$.

```ocaml
let r_sharp_1 = 14;;
let r1 = l1 r_sharp_1;;
let ww1 = mmod (q1_0 + q1_1 * r_sharp_1 + q1_2 * r_sharp_1 * r_sharp_1) modulus;;
```

3 Checking layer no. 2

Prover and verifier run Sum-Check for the functions $W_1$ and $W_2$, shown on slide 54 (page 57) of the slides.

Verifier sends the random value $r\#1$ to the Prover, this enables the Prover to learn $r_1$. The Prover has to convince him, that the summation in slide 54 (where the value of $X$ is $r_1$) is indeed equal to $ww_1$.

At the beginning of Sum-Check, verifier initializes the value $z_0$, which is the expected result of summation. This value is equal to $ww_1$ we defined above.

```ocaml
let z0_2 = ww1;;
```

3.1 Round 1 of Sum-Checking on layer 2

The prover sends to the verifier the coefficients of the polynomial on the right-hand side in slide 54, where the variable $X$ is at the position of the left bit of $b$.

```ocaml
let poly_2 = memo (function [x1;x2;x3;x4;x5;x6] ->
  let w2b = w2 [x3;x4]
  and w2c = w2 [x5;x6]
  and a = add_2 [x1;x2;x3;x4;x5;x6]
) x1 x2 x3 x4 x5 x6
```
and m = mult_2 [x1;x2;x3;x4;x5;x6]
  in
  mmod (a * (w2b + w2c) + m * w2b * w2c) modulus
| 1 -> raise (Failure ("poly_2 was called with " ^ (string_of_int (List.length _) - 1)) ^ " arguments!")

let (a0_21, a1_21, a2_21) = interp poly_2 6 r1;;

[28]: val poly_2 : int list -> int = <fun>
[28]: val a0_21 : int = 14
    val a1_21 : int = 12
    val a2_21 : int = 8

Verifier checks whether \( f(0) + f(1) = z0_2 \), where \( f(X) = a0_21 + a1_21 \cdot X + a2_21 \cdot X^2 \).
[29]: z0_2 = mmod (a0_21 + a1_21 + a2_21) modulus;;
[29]: ~ : bool = true

The verifier picks a random value \( r1 \) (try to change this value), and computes \( z1 \) as \( f(r1) \)
[30]: let r1_2 = 7;;
    let z1_2 = mmod (a0_21 + a1_21 * r1_2 + a2_21 * r1_2 * r1_2) modulus;;
[30]: val r1_2 : int = 7
[30]: val z1_2 : int = 14

Verifier sends the value of \( r1 \) to the prover.

3.2 Round 2 of Sum-Checking on layer 2

The prover sends to the verifier the coefficients of the polynomial on the right-hand side in slide 54, where the variable \( X \) is at the position of the right bit of \( b \).
[31]: let (a0_22,a1_22,a2_22) = interp poly_2 6 (r1 @ [r1_2]);;
[31]: val a0_22 : int = 11
    val a1_22 : int = 9
    val a2_22 : int = 0
Verifier checks whether \( f(0) + f(1) = z_{1,2} \), where \( f(X) = a_{0,22} + a_{1,22} \cdot X + a_{2,22} \cdot X^2 \).

\[
\begin{align*}
\textbf{[32]}: & \quad z_{1,2} = \text{mmod} (a_{0,22} + (a_{0,22} + a_{1,22} + a_{2,22})) \modulus; \\
\textbf{[32]}: & \quad - : \text{bool} = \text{true}
\end{align*}
\]

The verifier picks a random value \( r_2 \) (try to change this value), and computes \( z_2 \) as \( f(r_2) \)

\[
\begin{align*}
\textbf{[33]}: & \quad \text{let } r_2 = 5; \\
& \quad \text{let } z_2 = \text{mmod} (a_{0,22} + a_{1,22} \cdot r_2 + a_{2,22} \cdot r_2 \cdot r_2) \modulus; \\
\textbf{[33]}: & \quad \text{val } r_2 : \text{int} = 5 \\
\textbf{[33]}: & \quad \text{val } z_2 : \text{int} = 5
\end{align*}
\]

Verifier sends the value of \( r_2 \) to the prover.

### 3.3 Round 3 of Sum-Checking on layer 2

The prover sends to the verifier the coefficients of the polynomial on the right-hand side in slide 54, where the variable \( X \) is at the position of the left bit of \( c \).

\[
\begin{align*}
\textbf{[34]}: & \quad \text{let } (a_{0,23},a_{1,23},a_{2,23}) = \text{interp \ pol}_2 6 (r_1 @ [r_1;r_2]); \\
\textbf{[34]}: & \quad \text{val } a_{0,23} : \text{int} = 0 \\
& \quad \text{val } a_{1,23} : \text{int} = 9 \\
& \quad \text{val } a_{2,23} : \text{int} = 13
\end{align*}
\]

Verifier checks whether \( f(0) + f(1) = z_{2,2} \), where \( f(X) = a_{0,23} + a_{1,23} \cdot X + a_{2,23} \cdot X^2 \).

\[
\begin{align*}
\textbf{[35]}: & \quad z_{2,2} = \text{mmod} (a_{0,23} + (a_{0,23} + a_{1,23} + a_{2,23})) \modulus; \\
\textbf{[35]}: & \quad - : \text{bool} = \text{true}
\end{align*}
\]

The verifier picks a random value \( r_3 \) (try to change this value), and computes \( z_3 \) as \( f(r_3) \)

\[
\begin{align*}
\textbf{[36]}: & \quad \text{let } r_3 = 8; \\
& \quad \text{let } z_3 = \text{mmod} (a_{0,23} + a_{1,23} \cdot r_3 + a_{2,23} \cdot r_3 \cdot r_3) \modulus;
\end{align*}
\]
3.4 Round 4 of Sum-Checking on layer 2

The prover sends to the verifier the coefficients of the polynomial on the right-hand side in slide 54, where the variable X is at the position of the right bit of c.

```
let (a0_24, a1_24, a2_24) = interp poly_2 δ (r1 ⊗ [r1_2; r2_2; r3_2]);;
```

Verifier checks whether \( f(0) + f(1) = z3_2 \), where \( f(X) = a0_24 + a1_24 \cdot X + a2_24 \cdot X^2 \).

```
z3_2 = mmod (a0_24 + (a0_24 + a1_24 + a2_24)) modulus;;
```

```
- : bool = true
```

The verifier picks a random value \( r4 \) (try to change this value), and computes \( z4 \) as \( f(r4) \)

```
let r4_2 = 1;;
let z4_2 = mmod (a0_24 + a1_24 \cdot r4_2 + a2_24 \cdot r4_2 \cdot r4_2) modulus;;
```

```
val r4_2 : int = 1
val z4_2 : int = 0
```

3.5 Going from layer 2 to layer 3

Define the line \( l_2 \), going through the points \([r1_2; r2_2; r3_2; r4_2]\). The prover tells the verifier the coefficients of the polynomial \( q2 = w2 \cdot l_2 \), where the dot denotes composition. As \( w2 \)
takes two arguments, the degree of q2 is 2.

```ocaml
let l2 x = [mmod (r1_2 + (r3_2-r1_2) * x) modulus; mmod (r2_2 + (r4_2-r2_2) * x)] modulus];;

let (q2_0,q2_1,q2_2) = interpolate_parabola (w2 (l2 0)) (w2 (l2 1)) (w2 (l2 0)) (-1));;
```

Verifier can now complete the Sum-Check, using q2(0) and q2(1) as the values w2 [r1_2;r2_2] and w2 [r3_2;r4_2].

```ocaml
let w2b = q2_0
and w2c = mmod (q2_0 + q2_1 + q2_2) modulus
and a = add_2 (r1 @ [r1_2;r2_2;r3_2;r4_2])
and m = mult_2 (r1 @ [r1_2;r2_2;r3_2;r4_2])
in z4_2 = mmod (a * (w2b + w2c) + m * w2b * w2c) modulus;;
```

Verifier picks a random r#2 (try to change this value), defines r2 = l2(r#2) and ww2 = q2(r#2).

```ocaml
let r_sharp_2 = 10;;

let r2 = l2 r_sharp_2;;

let ww2 = mmod (q2_0 + q2_1 * r_sharp_2 + q2_2 * r_sharp_2 * r_sharp_2) modulus;;
```

```ocaml
val r_sharp_2 : int = 10

val r2 : int list = [0; 16]

val ww2 : int = 16
```
4 Checking layer no. 3

Prover and verifier run Sum-Check for the functions W2 and W3, shown on slide 54 (page 57) of the slides.

Verifier sends the random value r#2 to the Prover, this enables the Prover to learn r2. The Prover has to convince him, that the summation in slide 54 (where the value of X is r2) is indeed equal to ww2.

At the beginning of Sum-Check, verifier intializes the value z0, which is the expected result of summation. This value is equal to ww2 we defined above.

\[
\text{let } z0_3 = \text{ww2};
\]

\[
\text{val } z0_3 : \text{int} = 16
\]

4.1 Round 1 of Sum-Checking on layer 3

The prover sends to the verifier the coefficients of the polynomial on the right-hand side in slide 54, where the variable X is at the position of the left bit of b.

\[
\text{let } \text{poly}_3 = \text{memo (function } [x1;x2;x3;x4;x5;x6] \rightarrow
\text{let } w3b = w3 [x3;x4]
\text{and } w3c = w3 [x5;x6]
\text{and } a = \text{add}_3 [x1;x2;x3;x4;x5;x6]
\text{and } m = \text{mult}_3 [x1;x2;x3;x4;x5;x6]
\text{in }
\text{mmod (a } \ast \text{ (w3b + w3c) } + \text{ m } \ast \text{ w3b } \ast \text{ w3c) modulus }
\mid 1 \rightarrow \text{raise (Failure } \text{("poly}_3 \text{ was called with " } \sim \text{ (string_of_int (List.length_} \sim 1)) \sim \text{ " arguments!"})) \text{));}
\]

\[
\text{let } (a0_31, a1_31, a2_31) = \text{interp poly}_3 6 \text{ r2};
\]

\[
\text{val poly}_3 : \text{int list } \rightarrow \text{int} = \langle \text{fun} \rangle
\]

\[
\text{val } a0_31 : \text{int} = 16
\]
\[
\text{val } a1_31 : \text{int} = 10
\]
\[
\text{val } a2_31 : \text{int} = 8
\]

Verifier checks whether \( f(0) + f(1) = z0_3 \), where \( f(X) = a0_31 + a1_31 \ast X + a2_31 \ast X^2 \).

\[
\text{z0}_3 = \text{mmod (a0}_31 + (a0}_31 + a1)_31 + a2}_31)) \text{ modulus};
\]

\[
- : \text{bool} = \text{true}
\]
The verifier picks a random value \( r_1 \) (try to change this value), and computes \( z_1 \) as \( f(r_1) \)

\[
\text{let } r_1_3 = 4;; \\
\text{let } z_1_3 = \text{mmod} \ (a_{0_31} + a_{1_31} \cdot r_1_3 + a_{2_31} \cdot r_1_3 \cdot r_1_3) \modulus;;
\]

\[
\text{val } r_1_3 : \text{int} = 4
\]

\[
\text{val } z_1_3 : \text{int} = 14
\]

Verifier sends the value of \( r_1 \) to the prover.

### 4.2 Round 2 of Sum-Checking on layer 3

The prover sends to the verifier the coefficients of the polynomial on the right-hand side in slide 54, where the variable \( X \) is at the position of the right bit of \( b \).

\[
\text{let } (a_{0_32}, a_{1_32}, a_{2_32}) = \text{interp \ poly_3} \ 6 \ (r_2 @ \ [r_1_3]);;
\]

\[
\text{val } a_{0_32} : \text{int} = 14 \\
\text{val } a_{1_32} : \text{int} = 13 \\
\text{val } a_{2_32} : \text{int} = 7
\]

Verifier checks whether \( f(0) + f(1) = z_1_3 \), where \( f(X) = a_{0_32} + a_{1_32} \cdot X + a_{2_32} \cdot X^2 \).

\[
\text{z1_3 = mmod} \ (a_{0_32} + (a_{0_32} + a_{1_32} + a_{2_32})) \modulus;;
\]

\[
\text{- : bool} = \text{true}
\]

The verifier picks a random value \( r_2 \) (try to change this value), and computes \( z_2 \) as \( f(r_2) \)

\[
\text{let } r_2_3 = 2;; \\
\text{let } z_2_3 = \text{mmod} \ (a_{0_32} + a_{1_32} \cdot r_2_3 + a_{2_32} \cdot r_2_3 \cdot r_2_3 \cdot r_2_3) \modulus;;
\]

\[
\text{val } r_2_3 : \text{int} = 2
\]

\[
\text{val } z_2_3 : \text{int} = 0
\]

Verifier sends the value of \( r_2 \) to the prover.
4.3 Round 3 of Sum-Checking on layer 3

The prover sends to the verifier the coefficients of the polynomial on the right-hand side in slide 54, where the variable X is at the position of the left bit of c.

```
[50]: let (a0_33,a1_33,a2_33) = interp poly_3 6 (r2 @ [r1_3;r2_3]);;

[50]: val a0_33 : int = 7
val a1_33 : int = 11
val a2_33 : int = 9
```

Verifier checks whether \( f(0) + f(1) = z_2_3 \), where \( f(X) = a_{0_33} + a_{1_33} \cdot X + a_{2_33} \cdot X^2 \).

```
[51]: z2_3 = mmod (a0_33 + (a0_33 + a1_33 + a2_33)) modulus;;
[51]: - : bool = true
```

The verifier picks a random value \( r_3 \) (try to change this value), and computes \( z_3 \) as \( f(r_3) \)

```
[52]: let r3_3 = 4;;
    let z3_3 = mmod (a0_33 + a1_33 * r3_3 + a2_33 * r3_3 * r3_3) modulus;;
[52]: val r3_3 : int = 4
[52]: val z3_3 : int = 8
```

Verifier sends the value of \( r_3 \) to the prover.

4.4 Round 4 of Sum-Checking on layer 3

The prover sends to the verifier the coefficients of the polynomial on the right-hand side in slide 54, where the variable X is at the position of the right bit of c.

```
[53]: let (a0_34,a1_34,a2_34) = interp poly_3 6 (r2 @ [r1_3;r2_3;r3_3]);;

[53]: val a0_34 : int = 13
val a1_34 : int = 11
val a2_34 : int = 5
```

Verifier checks whether \( f(0) + f(1) = z_3_3 \), where \( f(X) = a_{0_34} + a_{1_34} \cdot X + a_{2_34} \cdot X^2 \).

```
[54]: z3_3 = mmod (a0_34 + (a0_34 + a1_34 + a2_34)) modulus;;
```
The verifier picks a random value $r_4$ (try to change this value), and computes $z_4$ as $f(r_4)$

```ml
let r4_3 = 15;;
let z4_3 = mmod (a0_34 + a1_34 * r4_3 + a2_34 * r4_3 * r4_3) modulus;;
```

```
val r4_3 : int = 15
val z4_3 : int = 11
```

The Verifier would like to evaluate $\text{poly}_3$ at the position $(r_1 @ [r_1_2;r_2_2;r_3_2;r_4_2])$. He can, as he is able to compute the values $w_3 [r_1_3;r_2_3]$ and $w_3 [r_3_3;r_4_3]$. Still, he does not want to evaluate $w_3$ twice, hence he get the Prover to help.

### 4.5 Finishing the last layer

Define the line $l_3$, going through the points $[r_1_3;r_2_3]$ and $[r_3_3;r_4_3]$. The prover tells the verifier the coefficients of the polynomial $q_3 = w_3 . l_3$, where the dot denotes composition. As $w_3$ takes two arguments, the degree of $q_3$ is 2.

```ml
let l3 x = [mmod (r1_3 + (r3_3-r1_3) * x) modulus; mmod (r2_3 + (r4_3-r2_3) * x) modulus];
let (q3_0,q3_1,q3_2) = interpolate_parabola (w3 (l3 0)) (w3 (l3 1)) (w3 (l3 (-1))));
```

```
val l3 : int -> int list = <fun>
```

```
val q3_0 : int = 1
val q3_1 : int = 7
val q3_2 : int = 0
```

Verifier can now complete the Sum-Check, using $q_3(0)$ and $q_3(1)$ as the values $w_3 [r_1_3;r_2_3]$ and $w_3 [r_3_3;r_4_3]$.

```ml
let w3b = q3_0
and w3c = mmod (q3_0 + q3_1 + q3_2) modulus
and a = add_3 (r2 @ [r1_3;r2_3;r3_3;r4_3])
and m = mult_3 (r2 @ [r1_3;r2_3;r3_3;r4_3])
in z4_3 = mmod (a * (w3b + w3c) + m * w3b * w3c) modulus;;
```
Verifier picks a random \( r^\#3 \) (try to change this value), defines \( r_3 = l_3(r^\#3) \) and \( w_3 = q_3(r^\#3) \).

```ocaml
let r_sharp_3 = 7;;
let r3 = l3 r_sharp_3;;
let w3 = mmod (q3_0 + q3_1 * r_sharp_3 + q3_2 * r_sharp_3 * r_sharp_3) modulus;;
```

Finally, verifier checks that \( w_3 \) is equal to \( w_3(r_3) \).

```ocaml
w3 = w3(r3);;
```

\[ - : \text{bool} = \text{true} \]