The Move Book

In programming languages, an expression is a unit of code that returns a value. In Move, almost everything is an expression, with the sole exception of the let statement, which is a declaration. In this section, we cover the types of expressions and introduce the concept of scope.

Expressions are sequenced with semicolons ; . If there's "no expression" after the semicolon, the compiler will insert a unit () , which represents an empty expression.

In the Primitive Types section, we introduced the basic types of Move. And to illustrate them, we used literals. A literal is a notation for representing a fixed value in source code. Literals can be used to initialize variables or directly pass fixed values as arguments to functions. Move has the following literals:

Arithmetic, logical, and bitwise operators are used to perform operations on values. Since these operations produce values, they are considered expressions.

A block is a sequence of statements and expressions enclosed in curly braces {} . It returns the value of the last expression in the block (note that this final expression must not have an ending semicolon). A block is an expression, so it can be used anywhere an expression is expected.

We go into detail about functions in the Functions section. However, we have already used function calls in previous sections, so it's worth mentioning them here. A function call is an expression that calls a function and returns the value of the last expression in the function body, provided the last expression does not have a terminating semi-colon.

Control flow expressions are used to control the flow of the program. They are also expressions, so they return a value. We cover control flow expressions in the Control Flow section. Here's a very brief overview:

Literals

In the Primitive Types section, we introduced the basic types of Move. And to illustrate them, we used literals. A literal is a notation for representing a fixed value in source code. Literals can be used to initialize variables or directly pass fixed values as arguments to functions. Move has the following literals:

```
bash let b = true; // true is a literal let n = 1000; // 1000 is a literal let h = 0x0A; // 0x0A is
a literal let v = b"hello"; // b"hello" is a byte vector literal let x = x"0A"; // x"0A" is a byte
vector literal let c = vector[1, 2, 3]; // vector[] is a vector literal
```

Arithmetic, logical, and bitwise operators are used to perform operations on values. Since these operations produce values, they are considered expressions.

```
bash let sum = 1 + 2; // 1 + 2 is an expression let sum = (1 + 2); // the same expression with
parentheses let is_true = true && false; // true && false is an expression let is_true = (true &&
false); // the same expression with parentheses
```

A block is a sequence of statements and expressions enclosed in curly braces {} . It returns the value of the last expression in the block (note that this final expression must not have an ending semicolon). A block is an expression, so it can be used anywhere an expression is expected.

``bash // block with an empty expression, however, the compiler will // insert an empty expression automatically: let none = { () }` // let none = {};

// block with let statements and an expression. let sum = { let a = 1; let b = 2; a + b // last expression is the value of the block };

// block is an expression, so it can be used in an expression and // doesn't have to be assigned to a variable. { let a = 1; let b = 2; a + b; // not returned - semicolon. // compiler automatically inserts an empty expression () }; ```

We go into detail about functions in the Functions section. However, we have already used function calls in previous sections, so it's worth mentioning them here. A function call is an expression that calls a function and returns the value of the last expression in the function body, provided the last expression does not have a terminating semi-colon.

```bash fun add(a: u8, b: u8): u8 { a + b }

# [test]
```

fun some_other() { let sum = add(1, 2); // not returned due to the semicolon. // compiler automatically inserts an empty expression `()` as return value of the block } ```

Control flow expressions are used to control the flow of the program. They are also expressions, so they return a value. We cover control flow expressions in the Control Flow section. Here's a very brief overview:

```bash // if is an expression, so it returns a value; if there are 2 branches, // the types of the branches must match. if (bool_expr) expr1 else expr2;

// while is an expression, but it returns `()`. while (bool_expr) { expr; };

// loop is an expression, but returns `()` as well. loop { expr; break }; ```

## Operators

Arithmetic, logical, and bitwise operators are used to perform operations on values. Since these operations produce values, they are considered expressions.

```
bash let sum = 1 + 2; // 1 + 2 is an expression let sum = (1 + 2); // the same expression with parentheses let is_true = true && false; // true && false is an expression let is_true = (true && false); // the same expression with parentheses
```

A block is a sequence of statements and expressions enclosed in curly braces {} . It returns the value of the last expression in the block (note that this final expression must not have an ending semicolon). A block is an expression, so it can be used anywhere an expression is expected.

```
``bash // block with an empty expression, however, the compiler will // insert an empty expression automatically
```
: let none = { () }` // let none = {};

// block with let statements and an expression. let sum = { let a = 1; let b = 2; a + b // last expression is the value of the block };

// block is an expression, so it can be used in an expression and // doesn't have to be assigned to a variable. { let a = 1; let b = 2; a + b; // not returned - semicolon. // compiler automatically inserts an empty expression `()` }; ```

We go into detail about functions in the Functions section. However, we have already used function calls in previous sections, so it's worth mentioning them here. A function call is an expression that calls a function and returns the value of the last expression in the function body, provided the last expression does not have a terminating semi-colon.

```bash fun add(a: u8, b: u8): u8 { a + b }

# [test]

fun some_other() { let sum = add(1, 2); // not returned due to the semicolon. // compiler automatically inserts an empty expression `()` as return value of the block } ```

Control flow expressions are used to control the flow of the program. They are also expressions, so they return a value. We cover control flow expressions in the Control Flow section. Here's a very brief overview:

```bash // if is an expression, so it returns a value; if there are 2 branches, // the types of the branches must match. if (bool_expr) expr1 else expr2;

// while is an expression, but it returns `()`. while (bool_expr) { expr; };

// loop is an expression, but returns `()` as well. loop { expr; break }; ```

## Blocks

A block is a sequence of statements and expressions enclosed in curly braces {} . It returns the value of the last expression in the block (note that this final expression must not have an ending semicolon). A block is an expression, so it can be used anywhere an expression is expected.

```
``bash // block with an empty expression, however, the compiler will // insert an empty expression automatically
```
: let none = { () }` // let none = {};
```

// block with let statements and an expression. let sum = { let a = 1; let b = 2; a + b // last expression is the value of the block };

// block is an expression, so it can be used in an expression and // doesn't have to be assigned to a variable. { let a = 1; let b = 2; a + b; // not returned - semicolon. // compiler automatically inserts an empty expression `()` }; ```

We go into detail about functions in the Functions section. However, we have already used function calls in previous sections, so it's worth mentioning them here. A function call is an expression that calls a function and returns the value of the last expression in the function body, provided the last expression does not have a terminating semi-colon.

```bash fun add(a: u8, b: u8): u8 { a + b }

# [test]

fun some_other() { let sum = add(1, 2); // not returned due to the semicolon. // compiler automatically inserts an empty expression `()` as return value of the block } ```

Control flow expressions are used to control the flow of the program. They are also expressions, so they return a value. We cover control flow expressions in the Control Flow section. Here's a very brief overview:

```bash // if is an expression, so it returns a value; if there are 2 branches, // the types of the branches must match. if (bool_expr) expr1 else expr2;

// while is an expression, but it returns `()`. while (bool_expr) { expr; };

// loop is an expression, but returns `()` as well. loop { expr; break }; ```

## Function Calls

We go into detail about functions in the Functions section. However, we have already used function calls in previous sections, so it's worth mentioning them here. A function call is an expression that calls a function and returns the value of the last expression in the function body, provided the last expression does not have a terminating semi-colon.

```bash fun add(a: u8, b: u8): u8 { a + b }

# [test]

fun some_other() { let sum = add(1, 2); // not returned due to the semicolon. // compiler automatically inserts an empty expression `()` as return value of the block } ```

Control flow expressions are used to control the flow of the program. They are also expressions, so they return a value. We cover control flow expressions in the Control Flow section. Here's a very brief overview:

```bash // if is an expression, so it returns a value; if there are 2 branches, // the types of the branches must match. if (bool_expr) expr1 else expr2;

// while is an expression, but it returns `()`. while (bool_expr) { expr; };

// loop is an expression, but returns `()` as well. loop { expr; break }; ```

## Control Flow Expressions

Control flow expressions are used to control the flow of the program. They are also expressions, so they return a value. We cover control flow expressions in the Control Flow section. Here's a very brief overview:

```bash // if is an expression, so it returns a value; if there are 2 branches, // the types of the branches must match. if (bool_expr) expr1 else expr2;

// while is an expression, but it returns `()`. while (bool_expr) { expr; };

// loop is an expression, but returns `()` as well. loop { expr; break }; ```
```