# Expressions

How to write Scribble expressions Scribble property annotations usually consist of three elements:

```

Copy ///

```

Example:

```

Copy /// #if_succeeds {:msg "This is an example"} a == 1;

```

The property type is one of several expected keywords (if_succeeds ,invariant ,if_updated ,assert ). Thepart defines a human readable label for the property. It is either in the format{:msg "The label"} or just the short-hand"The label" .

The condition is an expression that evaluates to a boolean (True / False). For the most part, you use the same syntax here as in regular Solidity expressions. To ease development we have added some special functions and operators that are described in this section.

Simple Expressions

You can use the following standard Solidity expressions in Scribble expressions:

1. Numbers, boolean litearls and addresse literals
2. Identifiers naming state variables and function arguments and return values
3. Member accesses (e.g.a.foo
4. wherea
5. is a struct)
6. Index accesses (e.g.a[5]
7. wherea
8. is an array or a map)
9. Unary operators --
10. and!
11. Binary operators -+
12. ,-
13. ,*
14. ,/
15. ,%
16. ,**
17. ,<<
18. ,>>
19. ,|
20. ,&
21. ,^
22. ,<
23. ,<=
24. ,>
25. ,>=
26. ,==
27. ,!=
28. ,&&
29. ,||
30. Ternary operators (e.g.x < y ? x : y
31. )
32. Function calls (e.g.foo(1,true, x)
33. )
34.

As of Solidity 0.8.0 all arithmetic ischecked by default - that is an exception is thrown (i.e. your code reverts) whenever an overflow or underflow happens. Scribble arithmetic on the other hand isunchecked - that is overflows/underflows would happen silently, without a revert. This is on purpose, as we want to minimize the conditions under which an exception would originate from the instrumentation. However this requires users to bemindful of whether overflow may happen in arithmetic inside their properties .

Changing State

Some Solidity expressions can change the state of a smart contract (e.g.i++ and (somewhat surprisingly)i=1 are both expressions modifying state in Solidity) . In Scribble we forbid such expressions, as we don't want annotations to change state in any way.

For the same reason, while Scribble expressions allow function calls, they only allow calls topure andview functions.

Scribble Language Features

We add several extensions to Scribble on top of the core Solidity language. These are described in this section.

Referring to the previous state

```

Copy syntax: old( )

```

You can't use old() in invariants! When writing properties, you'll often want to describe how state is expected tochange . For example, you might expect some persons balance toincrease .To write these properties, you'll be able to use theold() construct.

You can talk about state changing only inif_succeds andif_updated invariants. In the case ofif_succeeds invariantsold(expression) refers to the value of the expression before the function began executing. In the case ofif_updated old(expression) refers to the value of the expression before the annotated variable was updated.

Example 1: Constant Example 2: Ownership Example 3: Expression The balance of the sender before a transaction is equal to the balance after.

Scribble:

Copy old(balance[mgs.sender]) == balance[msg.sender]

``` The old owner is not the new owner

Scribble:

```

Copy old(owner) != owner

``` The sum of balances between the sender and receiver is the same before and after a transaction.

You can put any expression in old(...), not just single variables. ```

Copy old(balance[sender] + balance[receiver]) == balance[sender] + balance[receiver]

``` Note that the type of the expression insideold(expression) can be any primitive value type - e.g. number, fixed bytes, address, boolean, etc. However we disallow reference types such as arrays, maps, structs, strings and bytes. This is done to limit the overhead of our instrumentation, since making a copy of such (usually dynamically sized) types may require loops. In the future we amy reconsider this decision, given enough interest.

Binding return values

let := in

For convenience you can defined immutable variables that can be used inside of an annotation. This can be useful in several cases:

1. Reducing duplication. If the same expressionE appears multiple times inside of the annotationA , you can re-writeA aslet x := E in A' whereA' is the same asA withE replaced withx.
2. Unpacking multiple returns values of a function and giving them names (see below example)
3. Giving human-readable names to certain expressions/constats for more readability.

Function return values Binding constants Multiple lets In this example we assume there is a functionunpack() that returns three values: amount, sender and receiver.

In this example, we'll write a property that only uses one of the variables (the amount) and checks that it is bigger than zero.

If you don't use a variable replace it with_ to make your property easy to read. ```

Copy let amount, sender, _ := unpack(packedTransaction) in amout >= 0;

``` You are not limited to binding function return values.

Take the following expression for example, this evaluates to 25.

```

Copy let x := 5 in x * x

``` Sometimes you might need multiple let bindings.

In the following example we first bind thebalance variable, and then we use it to compute and bindhalfBalance . Which we end up using to checkhalfBalance >= amount .

Both balance and halfBalance can be used in the final expression! ```

Copy /// if_succeeds {:msg "You can only extract half of the current balance in one time"} /// let balance = token.balanceOf(this) in /// let halfBalance = balance / 2 in /// halfBalance >= amount; function extractFunds(uint amount) public returns (bool) { ... }

```

Let-bindings and old() expressions

Sometimes you may want to use alet- binding variable inside of anold() expression in the body of the let, as in the contrived example below:

```

Copy contract Foo { uint x;

/// #if_succeeds let t:= x in old(t+1) > 10; function foo() public { ... } }

```

However running the above code through scribble would result in the following error:

```

Copy tmp.sol:5:5 TypeError: Variable t is defined in the new context in (let t := x in (old((t + 1)) > 10)) but used in an old() expression

In:

if_succeeds let t:= x in old(t+1) > 10;

```

The issue we are running into is, that the expression thatt is bound to is implicitly computedafter the original function was called, but we are attempting to use it in a computation that is executedbefore the original function was called. Since doing this requires time travel, we obviously cannot make it happen. The fix in such a scenario is to wrap the expression thatt is bound to in anold() expression like so:

```

Copy /// #if_succeeds let t:= old(x) in old(t+1) > 10; function foo() public { ... }

```

If that doesn't fix our problem, then we should re-think the property we are trying to express.

Implication

==>

In our annotations, we'll often use implications to specify some kind of conditional logic (behaviours for example ).

An implication likeA ==> B describes thatB must be true ifA is true. IfA isn't true, then it doesn't matter whetherB is true or false. In natural language, you can usually describe an implication as:If A, then also B .

Fun Fact:A ==> B is equivalent to(!A || B) Example 1: Simple implication Example 2: Ownership If the input to a function is negative, then the output is as well.

Copy /// #if_succeeds {:msg "negativity"} input < 0 ==> output < 0; function convert(uint input) public returns (uint output) { ... }

``` If there is a change to the owner variable, then the sender was the old owner.

```

Copy owner != old(owner) ==> msg.sender == old(owner)

```

Unchecked sum

unchecked_sum()

Sometimes a property wants to talk about the sum of numeric values in an array or a map. For such cases we have introduced theunchecked_sum() builtin function. You can apply it to both numeric maps and arrays.

Example 1: Token balances Example 2: Constant Supply This example usesunchecked_sum() to ensure that the sum of balances is equal to themintedTokens variable.

```

Copy /// #invariant unchecked_sum(balances) == mintedTokens; contract Token { uint mintedTokens; mapping(address=>uint) balances; ... }

``` Here you can see how we use sum to make sure no new tokens get minted.

```

Copy /// #if_succeeds old(unchecked_sum(balances)) == unchecked_sum(balances); contract Token { mapping(address=>uint) balances; ... }

``` The return type forunchecked_sum() is always eitheruint256 orint256 (depending on whether the underlying map/array is signed). While it may seem confusing thatunchecked_sum(arr) isint256 even ifarr itself isint8 , this was done on purpose, to minimize the chance of overflow when working with small integer types.

Speaking of overflow, as the nameunchecked_sum suggests, the computation of the sum isUNCHECKED and thus mayoverflow silently . It is up to the user to add additional annotations to ensure that the sum of entries in an array doesn't overflow. In the future we may add a checked version of the sum, that fails loudly when the summation overflows.

Previous Hints Next Quantification Last updated2 years ago

Was this helpful?
```