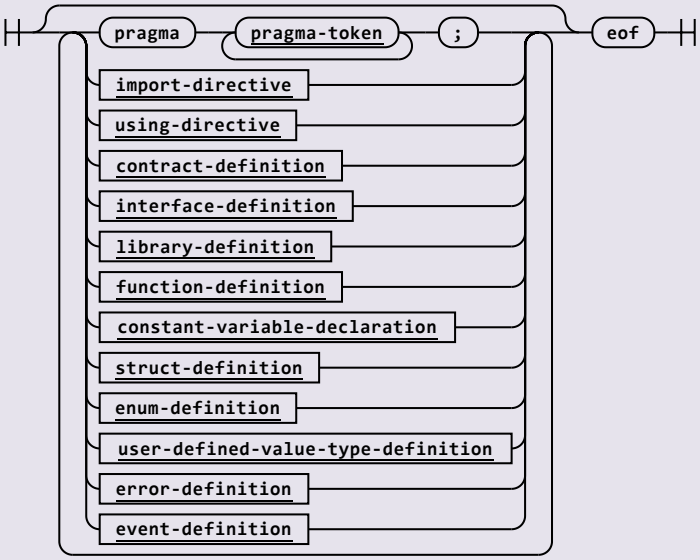
# Language Grammar

*parser grammar* **SolidityParser**

Solidity is a statically typed, contract-oriented, high-level language for implementing smart contracts on the Ethereum platform.

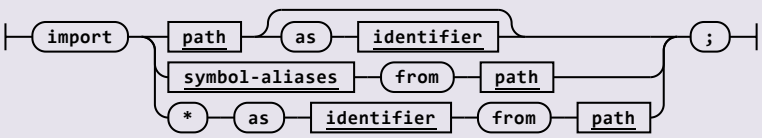
*rule* **source-unit**

On top level, Solidity allows pragmas, import directives, and definitions of contracts, interfaces, libraries, structs, enums and constants.




*rule* **import-directive**

Import directives import identifiers from different files.

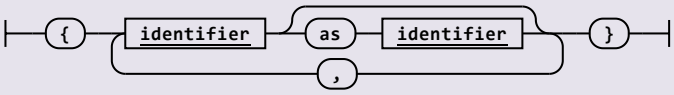



*rule* **path**

Path of a file to be imported.

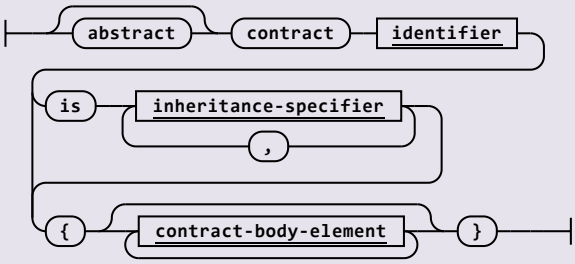
*rule* **symbol-aliases**

List of aliases for symbols to be imported.



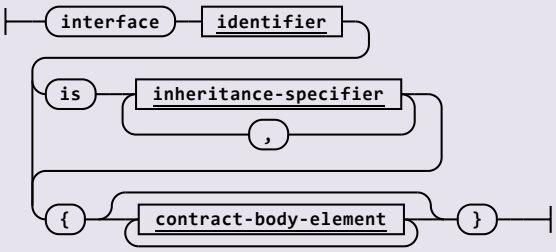

*rule* **contract-definition**

Top-level definition of a contract.



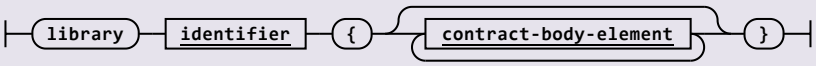

*rule* **interface-definition**

Top-level definition of an interface.




*rule* **library-definition**

Top-level definition of a library.



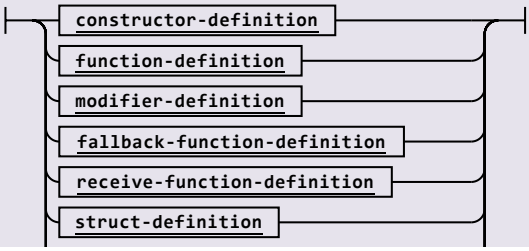

*rule* **inheritance-specifier**

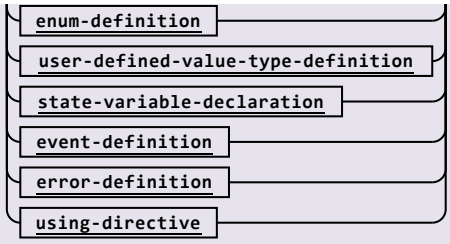
Inheritance specifier for contracts and interfaces. Can optionally supply base constructor arguments.


*rule* **contract-body-element**

Declarations that can be used in contracts, interfaces and libraries.

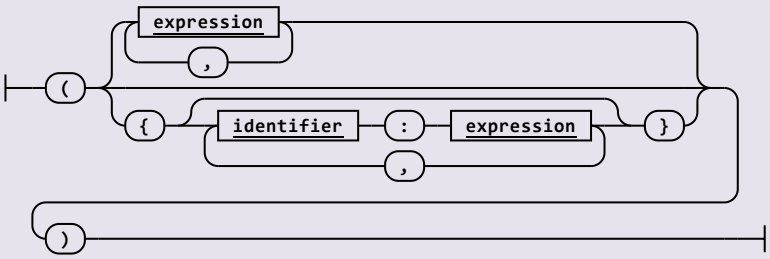
Note that interfaces and libraries may not contain constructors, interfaces may not contain state variables and libraries may not contain fallback, receive functions nor non-constant state variables.






*rule* **call-argument-list**

Arguments when calling a function or a similar callable object. The arguments are either given as comma separated list or as map of named arguments.




*rule* **identifier-path**

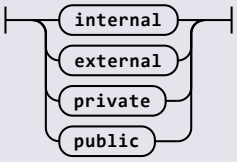
Qualified name.


*rule* **modifier-invocation**

Call to a modifier. If the modifier takes no arguments, the argument list can be skipped entirely (including opening and closing parentheses).

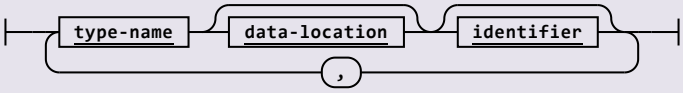

*rule* **visibility**

Visibility for functions and function types.



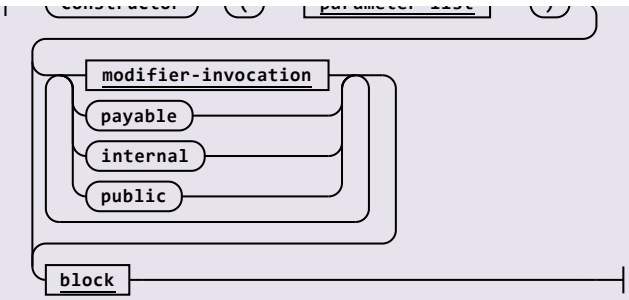

*rule* **parameter-list**

A list of parameters, such as function arguments or return values.



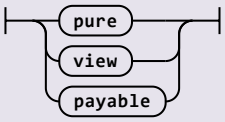

*rule* **constructor-definition**

Definition of a constructor. Must always supply an implementation. Note that specifying internal or public visibility is deprecated.




*rule* **state-mutability**

State mutability for function types. The default mutability 'non-payable' is assumed if no mutability is specified.

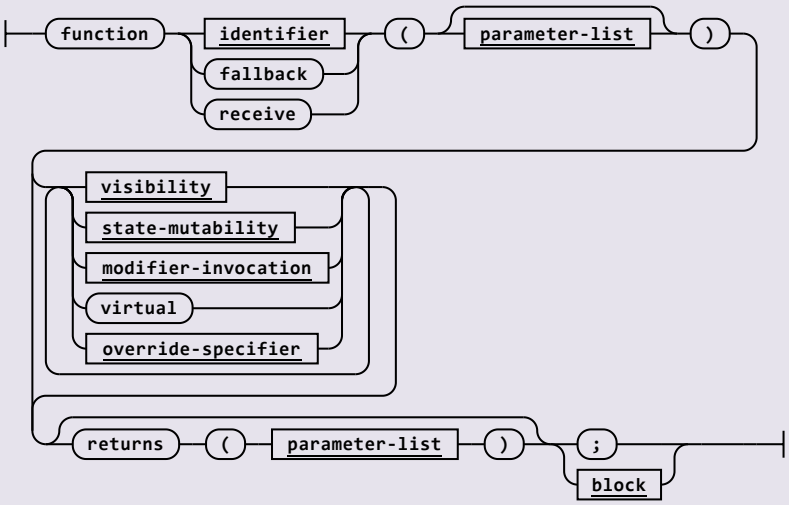



*rule* **override-specifier**

An override specifier used for functions, modifiers or state variables. In cases where there are ambiguous declarations in several base contracts being overridden, a complete list of base contracts has to be given.


*rule* **function-definition**

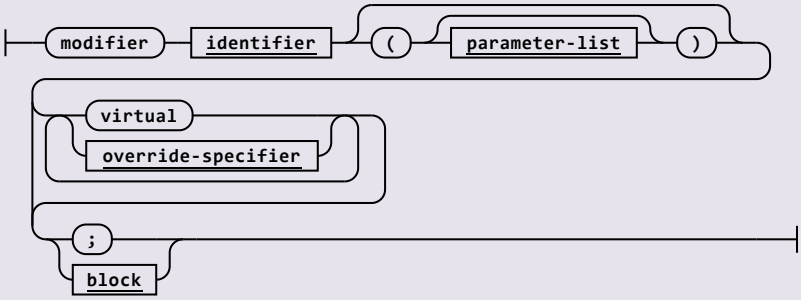
The definition of contract, library, interface or free functions. Depending on the context in which the function is defined, further restrictions may apply, e.g. functions in interfaces have to be unimplemented, i.e. may not contain a body block.




*rule* **modifier-definition**

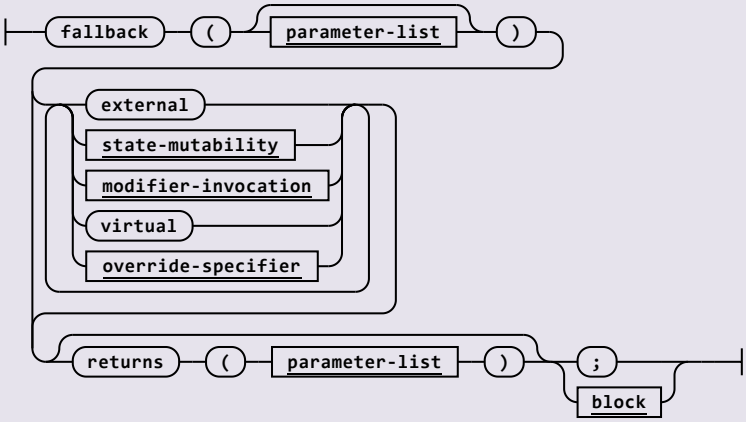
The definition of a modifier. Note that within the body block of a modifier, the underscore cannot be used as identifier, but is used as placeholder statement for the body of a

function to which the modifier is applied.



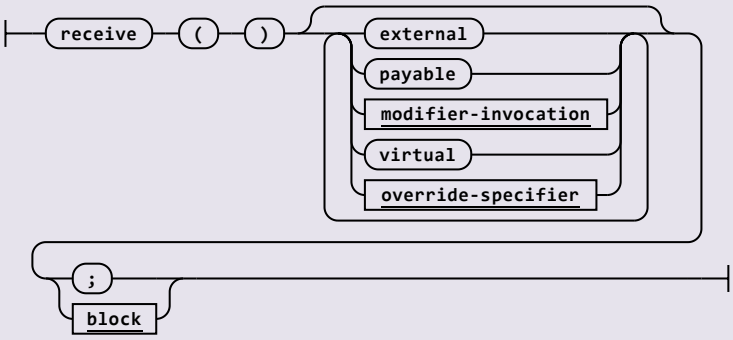

*rule* **fallback-function-definition**

Definition of the special fallback function.



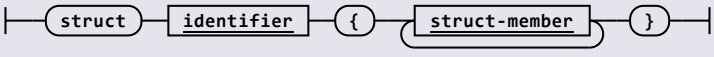

*rule* **receive-function-definition**

Definition of the special receive function.



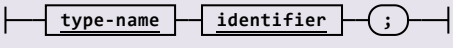

*rule* **struct-definition**

Definition of a struct. Can occur at top-level within a source unit or within a contract, library or interface.



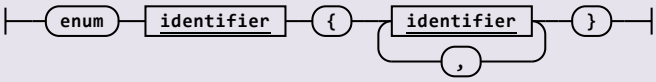

*rule* **struct-member**

The declaration of a named struct member.



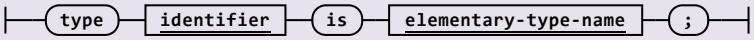

*rule* **enum-definition**

Definition of an enum. Can occur at top-level within a source unit or within a contract, library or interface.



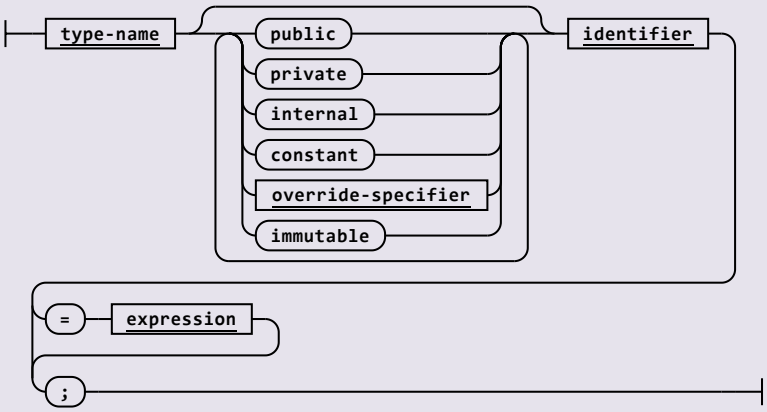

*rule* **user-defined-value-type-definition**

Definition of a user defined value type. Can occur at top-level within a source unit or within a contract, library or interface.




*rule* **state-variable-declaration**

The declaration of a state variable.




*rule* **constant-variable-declaration**

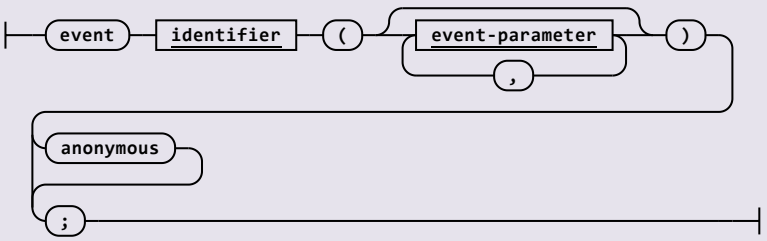
The declaration of a constant variable.


*rule* **event-parameter**

Parameter of an event.

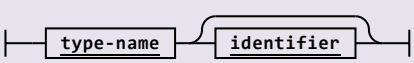

*rule* **event-definition**

Definition of an event. Can occur in contracts, libraries or interfaces.



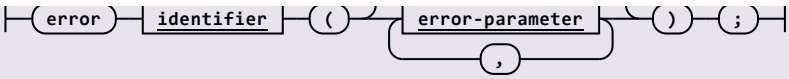

*rule* **error-parameter**

Parameter of an error.



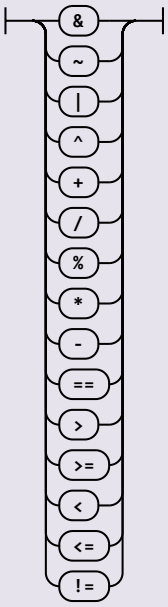

*rule* **error-definition**

Definition of an error.



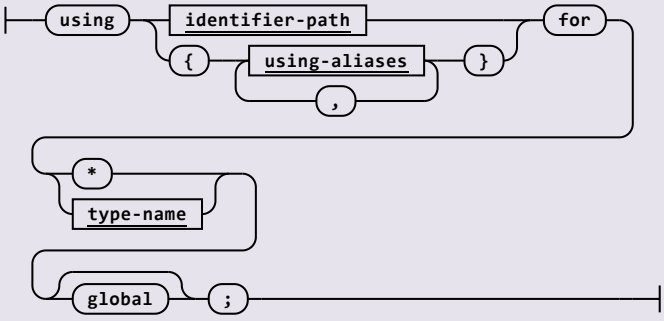

*rule* **user-definable-operator**

Operators that users are allowed to implement for some types with *using for*.

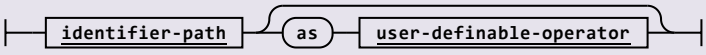



*rule* **using-directive**

Using directive to attach library functions and free functions to types. Can occur within contracts and libraries and at the file level.

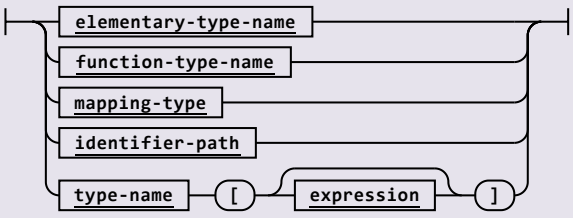



*rule* **using-aliases**

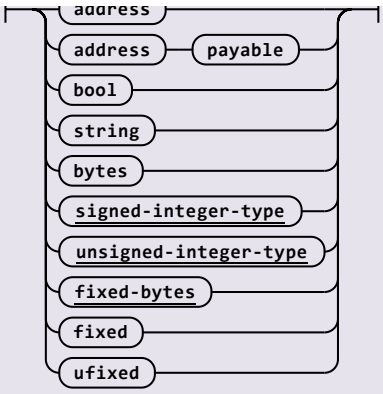



*rule* **type-name**

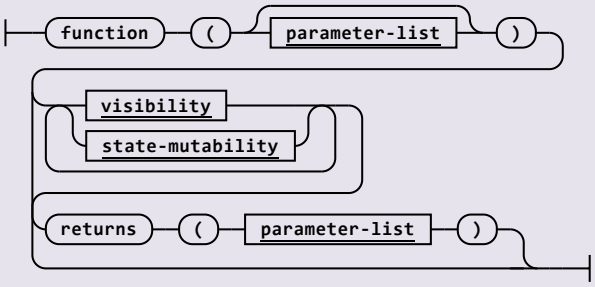
A type name can be an elementary type, a function type, a mapping type, a user-defined type (e.g. a contract or struct) or an array type.




*rule* **elementary-type-name**

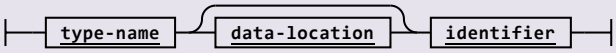



*rule* **function-type-name**

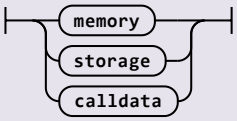



*rule* **variable-declaration**

The declaration of a single variable.

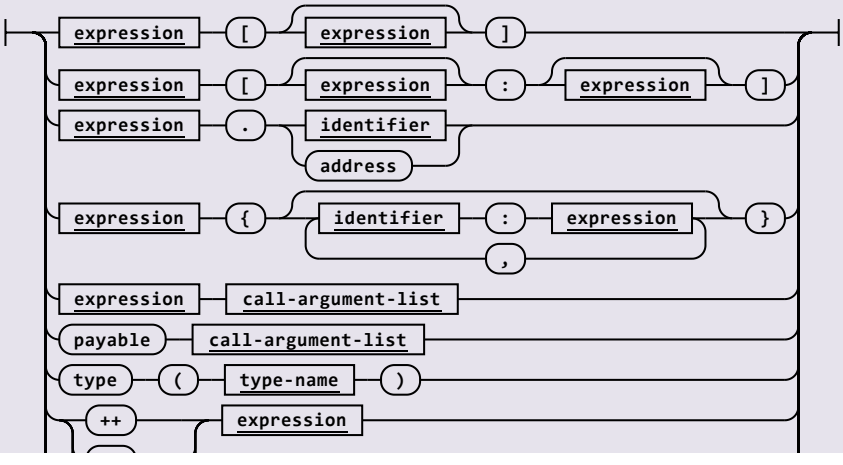


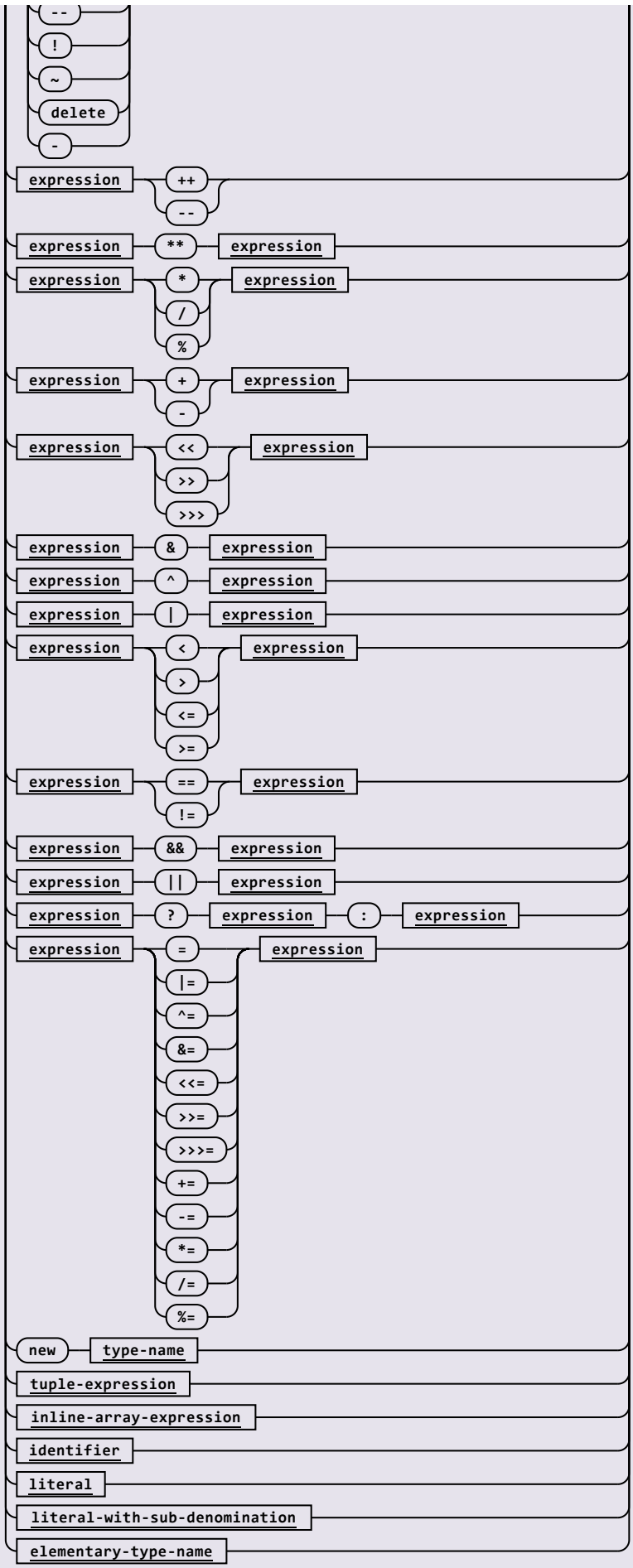

*rule* **data-location**




*rule* **expression**

Complex expression. Can be an index access, an index range access, a member access, a function call (with optional function call options), a type conversion, an unary or binary expression, a comparison or assignment, a ternary expression, a new-expression (i.e. a contract creation or the allocation of a dynamic memory array), a tuple, an inline array or a primary expression (i.e. an identifier, literal or type name).

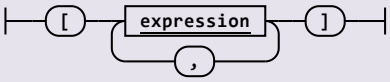





*rule* tuple-expression

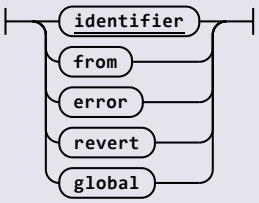

*rule* **inline-array-expression**

An inline array expression denotes a statically sized array of the common type of the contained expressions.

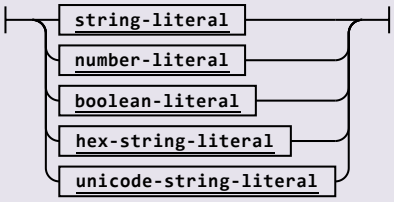



*rule* **identifier**

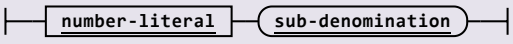
Besides regular non-keyword Identifiers, some keywords like ‘from’ and ‘error’ can also be used as identifiers.



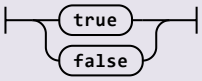

*rule* **literal**




*rule* **literal-with-sub-denomination**

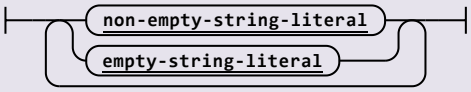



*rule* **boolean-literal**



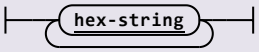

*rule* **string-literal**

A full string literal consists of either one or several consecutive quoted strings.



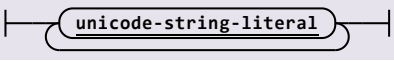

*rule* **hex-string-literal**

A full hex string literal that consists of either one or several consecutive hex strings.



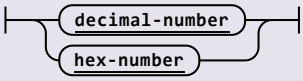

*rule* **unicode-string-literal**

A full unicode string literal that consists of either one or several consecutive unicode strings.



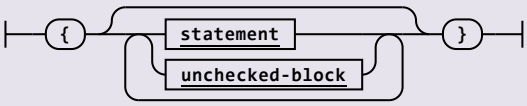
*rule* **number-literal**

Number literals can be decimal or hexadecimal numbers with an optional unit.

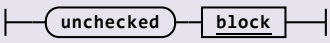



*rule* **block**

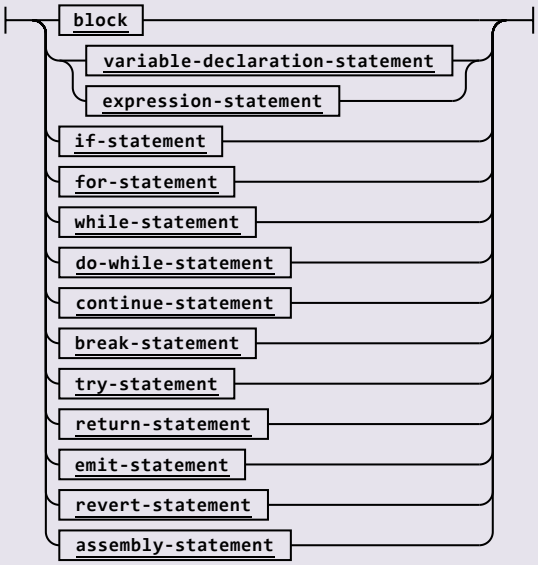
A curly-braced block of statements. Opens its own scope.




*rule* **unchecked-block**

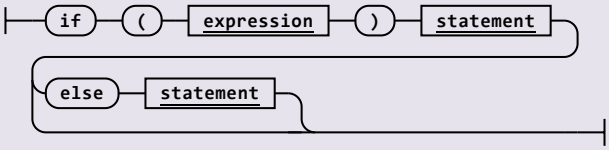



*rule* **statement**



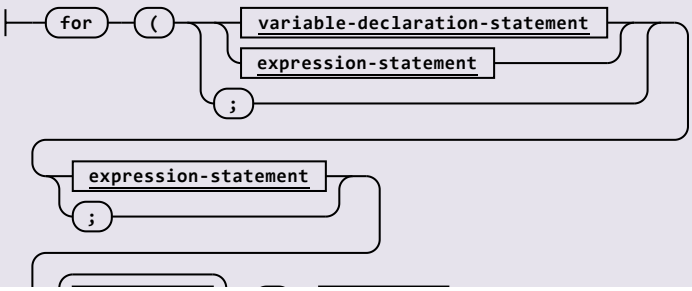

*rule* **if-statement**

If statement with optional else part.



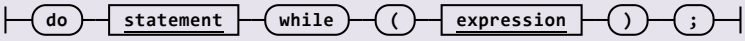

*rule* **for-statement**

For statement with optional init, condition and post-loop part.



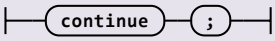

*rule* **while-statement**


*rule* **do-while-statement**




*rule* **continue-statement**

A continue statement. Only allowed inside for, while or do-while loops.

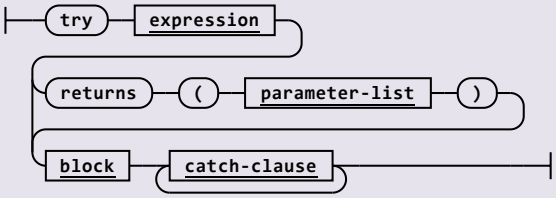



*rule* **break-statement**

A break statement. Only allowed inside for, while or do-while loops.

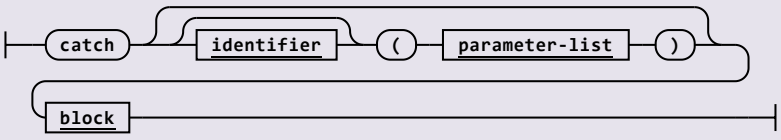

*rule* **try-statement**

A try statement. The contained expression needs to be an external function call or a contract creation.

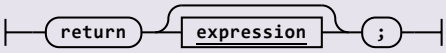



*rule* **catch-clause**

The catch clause of a try statement.

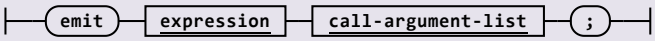



*rule* **return-statement**



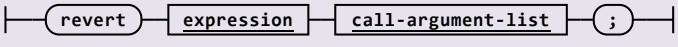

*rule* **emit-statement**

An emit statement. The contained expression needs to refer to an event.



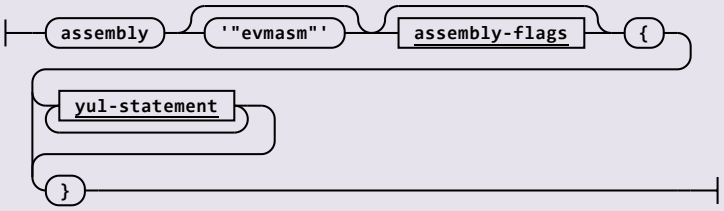

*rule* **revert-statement**

A revert statement. The contained expression needs to refer to an error.



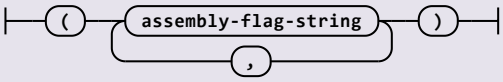
*rule* **assembly-statement**

An inline assembly block. The contents of an inline assembly block use a separate scanner/lexer, i.e. the set of keywords and allowed identifiers is different inside an inline assembly block.



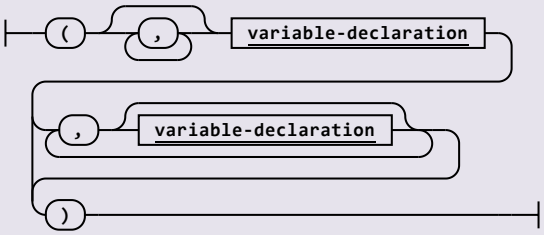

*rule* **assembly-flags**

Assembly flags. Comma-separated list of double-quoted strings as flags.



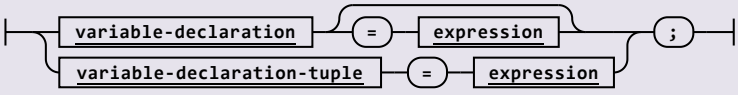

*rule* **variable-declaration-tuple**

A tuple of variable names to be used in variable declarations. May contain empty fields.

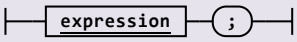



*rule* **variable-declaration-statement**

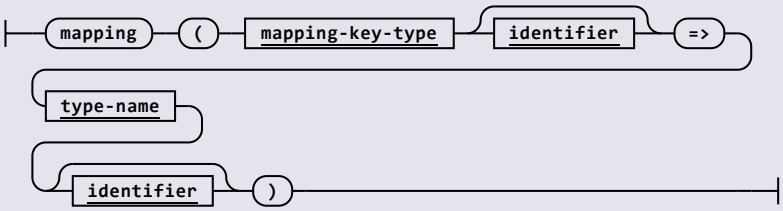
A variable declaration statement. A single variable may be declared without initial value, whereas a tuple of variables can only be declared with initial value.




*rule* **expression-statement**

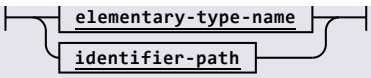



*rule* **mapping-type**



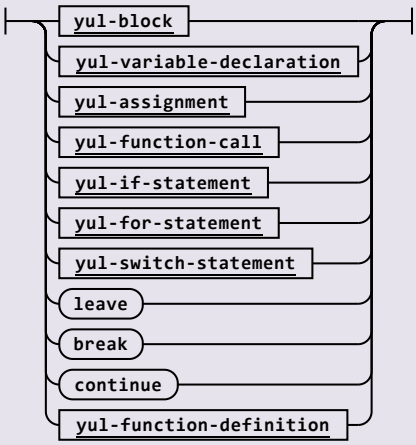

*rule* **mapping-key-type**

Only elementary types or user defined types are viable as mapping keys.

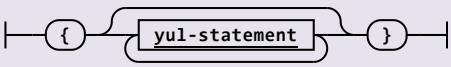



*rule* **yul-statement**

A Yul statement within an inline assembly block. continue and break statements are only valid within for loops. leave statements are only valid within function bodies.

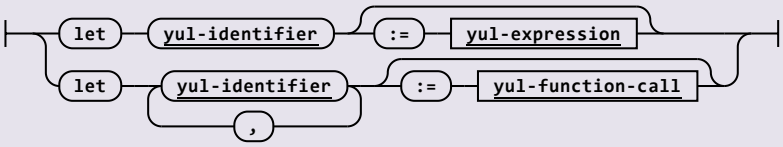



*rule* **yul-block**



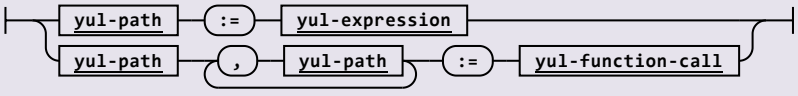

*rule* **yul-variable-declaration**

The declaration of one or more Yul variables with optional initial value. If multiple variables are declared, only a function call is a valid initial value.

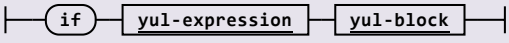



*rule* **yul-assignment**

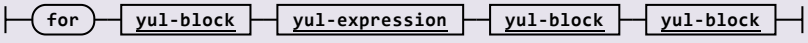
Any expression can be assigned to a single Yul variable, whereas multi-assignments require a function call on the right-hand side.




*rule* **yul-if-statement**



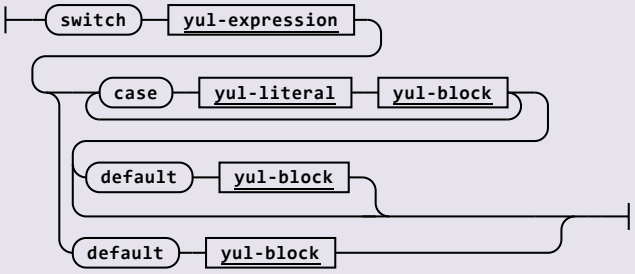

*rule* **yul-for-statement**



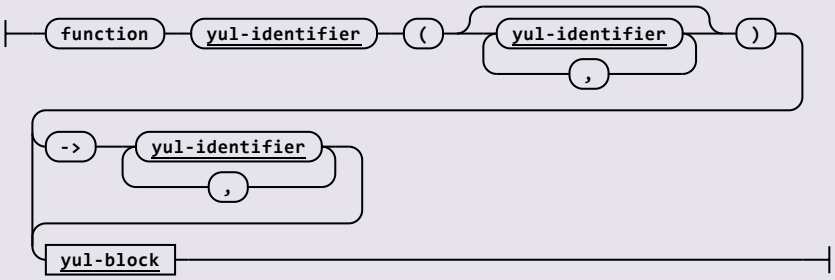

*rule* **yul-switch-statement**

A Yul switch statement can consist of only a default-case (deprecated) or one or more non-default cases optionally followed by a default-

case.

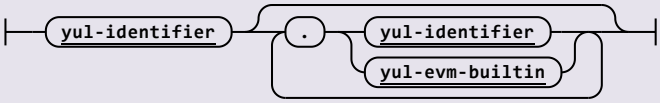



*rule* **yul-function-definition**



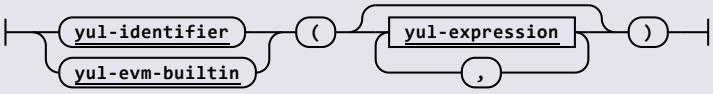

*rule* **yul-path**

While only identifiers without dots can be declared within inline assembly, paths containing dots can refer to declarations outside the inline assembly block.

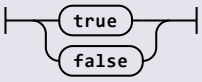



*rule* **yul-function-call**

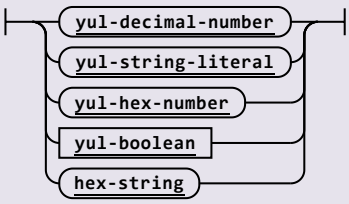
A call to a function with return values can only occur as right-hand side of an assignment or a variable declaration.



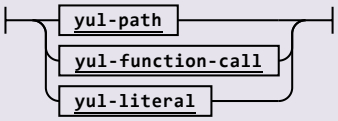

*rule* **yul-boolean**




*rule* **yul-literal**



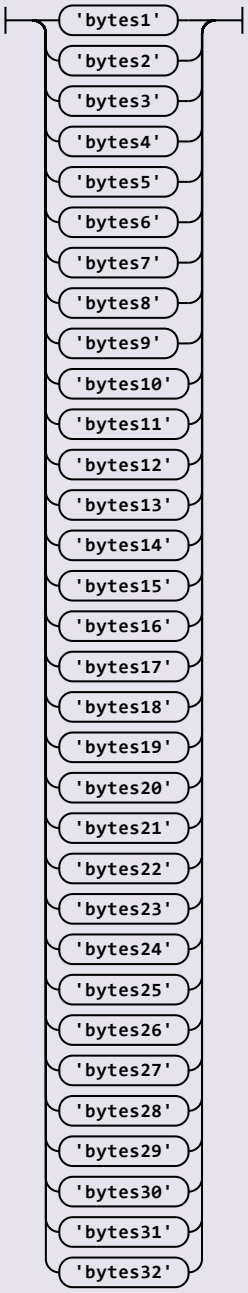

*rule* **yul-expression**



## *lexer grammar* SolidityLexer

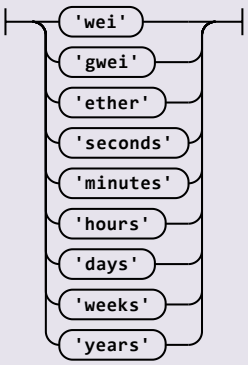
### *rule* fixed-bytes

Bytes types of fixed length.



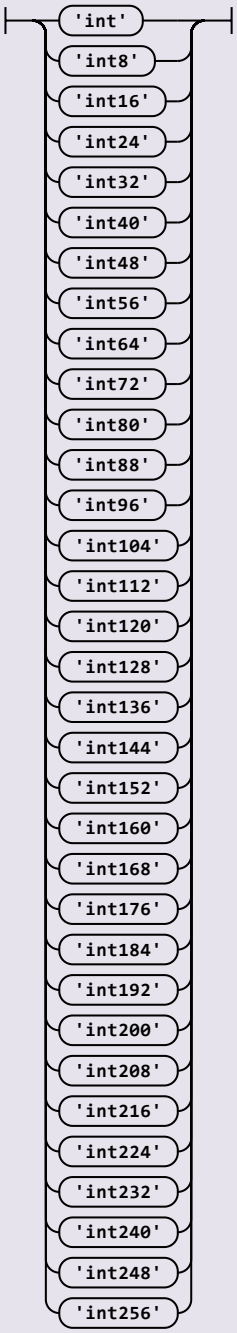

### *rule* sub-denomination

Unit denomination for numbers.



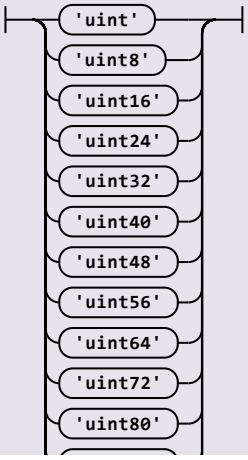

### *rule* signed-integer-type

Sized signed integer types. int is an alias of int256.



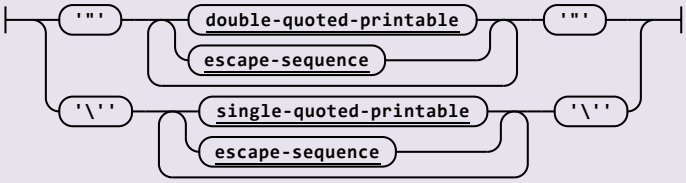

*rule* **unsigned-integer-type**

Sized unsigned integer types. uint is an alias of uint256.



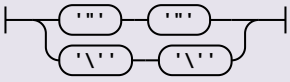

*rule* **non-empty-string-literal**

A non-empty quoted string literal restricted to printable characters.



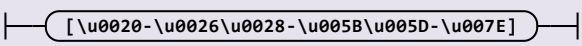

*rule* **empty-string-literal**

An empty string literal



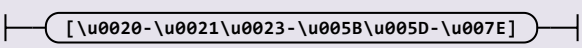

*rule* **single-quoted-printable**

Any printable character except single quote or back slash.




*rule* **double-quoted-printable**

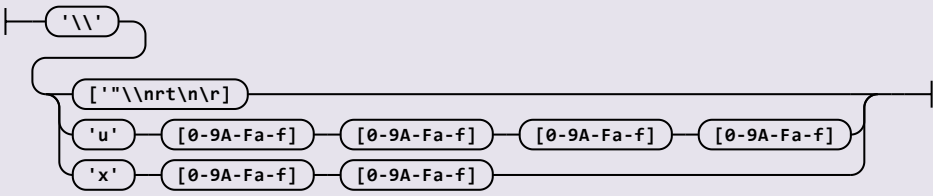
Any printable character except double quote or back slash.




*rule* **escape-sequence**

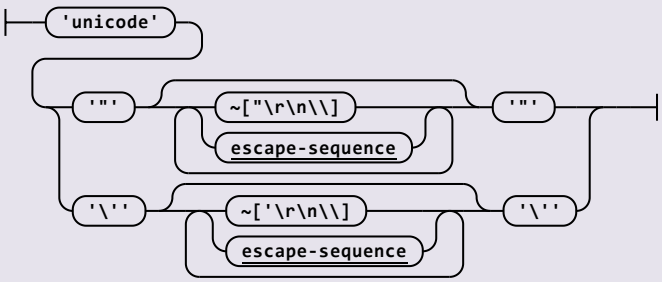
Escape sequence. Apart from common single character escape sequences, line breaks can be

escaped as well as four hex digit unicode escapes \uXXXX and two digit hex escape sequences \xXX are allowed.



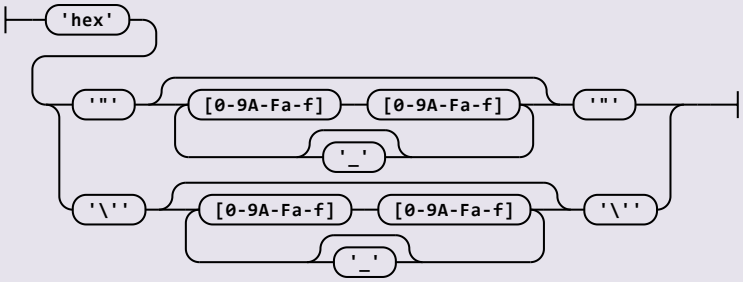

*rule* **unicode-string-literal**

A single quoted string literal allowing arbitrary unicode characters.



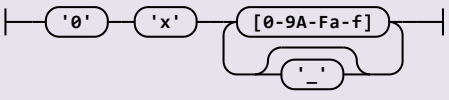

*rule* **hex-string**

Hex strings need to consist of an even number of hex digits that may be grouped using underscores.



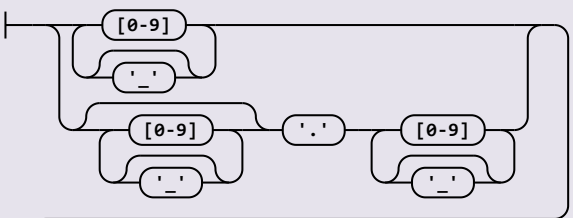

*rule* **hex-number**

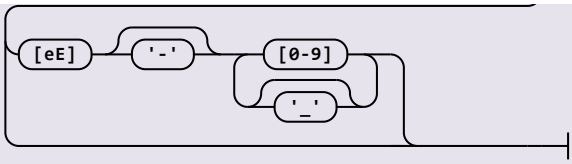
Hex numbers consist of a prefix and an arbitrary number of hex digits that may be delimited by underscores.




*rule* **decimal-number**

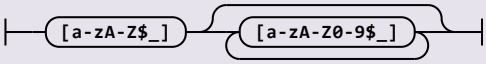
A decimal number literal consists of decimal digits that may be delimited by underscores and an optional positive or negative exponent. If the digits contain a decimal point, the literal has fixed point type.





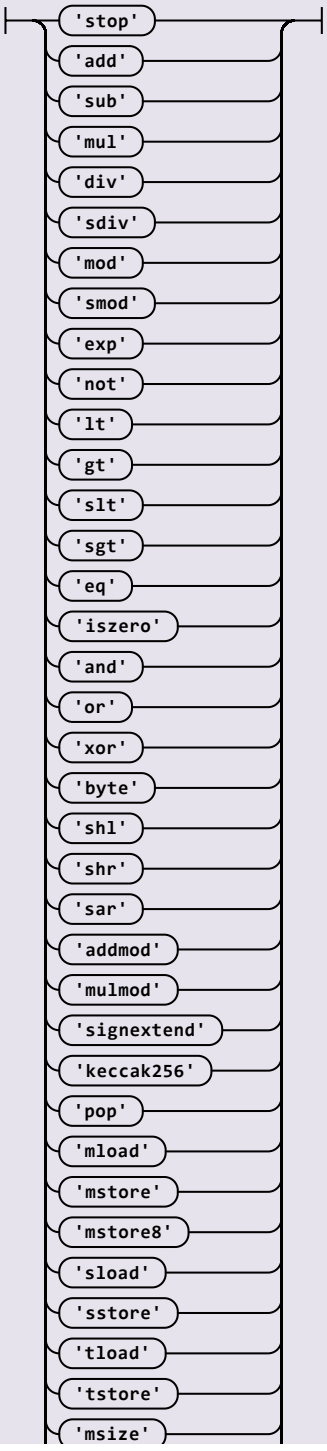

*rule* **identifier**

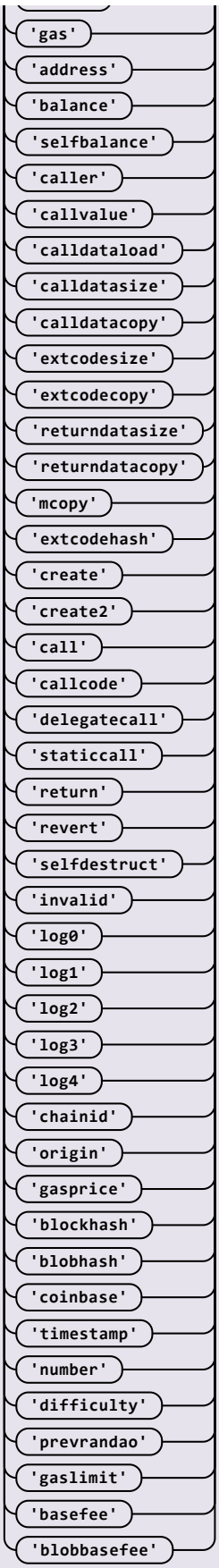
An identifier in solidity has to start with a letter, a dollar-sign or an underscore and may additionally contain numbers after the first symbol.




*rule* **yul-evm-builtin**

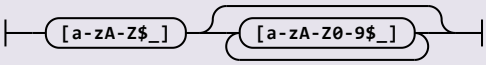
Builtin functions in the EVM Yul dialect.





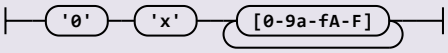

*rule* **yul-identifier**

Yul identifiers consist of letters, dollar signs, underscores and numbers, but may not start with a number. In inline assembly there cannot be dots in user-defined identifiers. Instead see yulPath for expressions consisting of identifiers with dots.



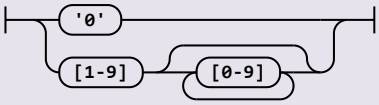

*rule* **yul-hex-number**

Hex literals in Yul consist of a prefix and one or more hexadecimal digits.



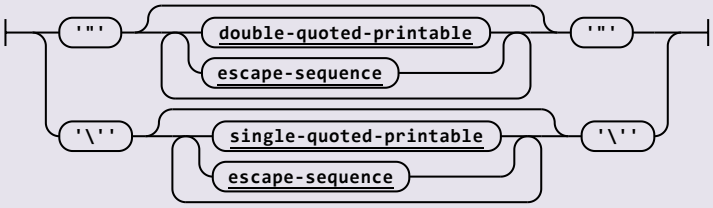

*rule* **yul-decimal-number**

Decimal literals in Yul may be zero or any sequence of decimal digits without leading zeroes.




*rule* **yul-string-literal**

String literals in Yul consist of one or more double-quoted or single-quoted strings that may contain escape sequences and printable characters except unescaped line breaks or unescaped double-quotes or single-quotes, respectively.




*rule* **pragma-token**

Pragma token. Can contain any kind of symbol except a semicolon. Note that currently the solidity parser only allows a subset of this.

Previous Next