5.2 Algorithm conventions

We often use a numbered list to specify steps in an algorithm. These algorithms are used to clarify semantics. In practice, there may be more efficient algorithms available to implement a given feature.

When an algorithm is to produce a value as a result, we use the directive return x to indicate that the result of the algorithm is the value of x and that the algorithm should terminate. We use the notation Result(n) as shorthand for the result of step n. We also use Type(x) as shorthand for the type of x.

Mathematical operations such as addition, subtraction, negation, multiplication, division, and the mathematical functions defined later in this section should always be understood as computing exact mathematical results on mathematical real numbers, which do not include infinities and do not include a negative zero that is distinguished from positive zero. Algorithms in this standard that model floating-point arithmetic include explicit steps, where necessary, to handle infinities and signed zero and to perform rounding. If a mathematical operation or function is applied to a floating-point number, it should be understood as being applied to the exact mathematical value represented by that floating-point number. Such a floating-point number must be finite, and if it is +0 or −0 then the corresponding mathematical value is simply 0.

The mathematical function \( \text{abs}(x) \) yields the absolute value of x, which is \(-x\) if x is negative (less than zero) and otherwise is x itself.

The mathematical function \( \text{sign}(x) \) yields 1 if x is positive and \(-1\) if x is negative. The sign function is not used in this standard for cases when x is zero.

The notation \( x \mod y \) (y must be finite and nonzero) computes a value \( k \) of the same sign as y such that \( \text{abs}(k) < \text{abs}(y) \) and \( x - k = q \cdot y \) for some integer \( q \).

The mathematical function floor(x) yields the largest integer (closest to positive infinity) that is not larger than x. Note that \( \text{floor}(x) = x - (x \mod 1) \).

If an algorithm is defined to generate a runtime error, execution of the algorithm is terminated and no result is returned. The calling algorithms are also terminated, until an algorithm step is reached that explicitly deals with the error. The same applies for exceptions that are explicitly thrown. See section 12.1. The algorithm step that deals with the runtime error, or the explicitly thrown exception, has available to it the details about the error, or the value thrown by the throw statement, respectively.

8.9 The Completion Type

The internal Completion type is not a language data type. It is defined by this specification purely for expository purposes. An implementation of ECMAScript must behave as if it produced and operated upon Completion values in the manner described here. However, a value of the Completion type is used only as an intermediate result of statement evaluation and cannot be stored as the value of a variable or property.

The Completion type is used to explain the behavior of statements (break, continue, return and throw) that perform nonlocal transfers of control. Values of the Completion type are triples of the form (type, value, target), where type is one of normal, break, continue, return, or throw, value is any ECMAScript value, or empty, and target is any ECMAScript identifier, or empty. If C is a completion triple, then the notation C.type denotes the first element, C.value the second and C.target the third.

The term “abrupt completion” refers to any completion with a reason value type other than normal.

Invoking the [[Call]] or [[Construct]] method of a Function object, amounts to the evaluation of a Block (see section 12.1) in an appropriate Execution Context (see section 10). The result of evaluating a Block is of the Completion Type. This value should not be returned as the result of the method invocation, or it might end up being stored in a variable or property. Instead, the value field of the completion value becomes the result of the invocation, except that an empty value is replaced with undefined. If the completion value is of type throw, execution of the algorithm that invoked the method should proceed as if a runtime error has occurred, see section 5.2.

12 Statements

Syntax
Statement:
Block
FunctionDeclaration
VariableDeclaration
EmptyStatement
ExpressionStatement
IfStatement
IterationStatement
ContinueStatement
BreakStatement
ReturnStatement
WithStatement
LabeledStatement
SwitchStatement
ThrowStatement
TryStatement

Semantics
A Statement can be part of a LabeledStatement, which itself can be part of a LabeledStatement, and so on. The labels introduced this way are collectively referred to as the “current label set” when describing the semantics of individual statements. A LabeledStatement has no semantic meaning other than the introduction of a label to a label set. An IterationStatement, or SwitchStatement that is not part of a LabeledStatement is regarded as possessing a label set containing a single element, empty.

12.1 Block
Syntax
Block:
    { StatementListopt }

StatementList:
    Statement
    StatementList Statement

Semantics
The production Block: { } is evaluated as follows:
1. Return (normal, empty, empty).

The production Block: { StatementList } is evaluated as follows:
1. Evaluate StatementList.
2. Return Result(1).

The production StatementList: Statement is evaluated as follows:
1. Evaluate Statement.
2. If an exception value was thrown during the evaluation of Statement, go to step 7.
3. If a runtime error occurred during the evaluation of Statement, go to step 5.
4. Return Result(1).
5. Construct an appropriate Error object.
7. Return (throw, V, empty) where V is the exception value thrown during the evaluation of Statement.

The production StatementList: StatementList Statement is evaluated as follows:
1. Evaluate StatementList.
2. If Result(1).type = break and Result(1).target occurs in the current label set, return (normal, Result(1).value, empty).
3. If Result(1) is an abrupt completion, return Result(1).
4. Evaluate Statement.
5. If Result(1).value = empty, let V = Result(1).value, otherwise let V = Result(4).value.

If an exception value was thrown during the evaluation of Statement, go to step 10.
8. If a runtime error occurred during the evaluation of Statement, go to step 8.
9. Return (throw, Result(8), empty).
10. Return (throw, W, empty) where W is the exception value thrown during the evaluation of Statement.

12.2 Variable statement

Syntax

VariableStatement : var VariableDeclarationList ;

VariableDeclarationList : VariableDeclaration VariableDeclarationList , VariableDeclaration

VariableDeclaration : Identifier Initializer opt

Initializer : = AssignmentExpression

Description

If the variable statement occurs inside a FunctionDeclaration, the variables are defined with function-local scope in that function, as described in section 10.1.3. Otherwise, they are defined with global scope, that is, they are created as members of the global object, as described in section 10.1.6. Variables are created when the execution scope is entered. A Block does not define a new execution scope. Only Program and FunctionDeclaration produce a new scope. Variables are initialized to the undefined value when created. A variable with an Initializer is assigned the value of its AssignmentExpression when the VariableStatement is executed, not when the variable is created.

Semantics

The production VariableStatement : var VariableDeclarationList ; is evaluated as follows:
1. Evaluate VariableDeclarationList.
2. Return (normal, empty, empty).

The production VariableDeclarationList : VariableDeclaration is evaluated as follows:
1. Evaluate VariableDeclaration.

The production VariableDeclarationList : VariableDeclarationList , VariableDeclaration is evaluated as follows:
1. Evaluate VariableDeclarationList.
2. Evaluate VariableDeclaration.

The production VariableDeclaration : Identifier is evaluated as follows:
1. Return a string value containing the same sequence of characters as in the Identifier.

The production VariableDeclaration : Identifier Initializer is evaluated as follows:
1. Evaluate Identifier.
2. Evaluate Initializer.
3. Call GetValue(Result(2)).
4. Call PutValue(Result(1), Result(3)).
5. Return a string value containing the same sequence of characters as in the Identifier.

The production \texttt{Initializer := AssignmentExpression} is evaluated as follows:

1. Evaluate AssignmentExpression.
2. Return Result(1).

### 12.3 Empty statement

**Syntax**

\texttt{EmptyStatement : ;}

**Semantics**

The production \texttt{EmptyStatement : ;} is evaluated as follows:

1. Return (\texttt{normal, empty, empty}).

### 12.4 Expression statement

**Syntax**

\texttt{ExpressionStatement : Expression ;}

**Semantics**

The production \texttt{ExpressionStatement : Expression ;} is evaluated as follows:

1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Return (\texttt{normal, Result(2), empty}).

### 12.5 The \texttt{if} statement

**Syntax**

\texttt{IfStatement : if ( Expression ) Statement else Statement}

\texttt{if ( Expression ) Statement}

Each \texttt{else} for which the choice of associated \texttt{if} is ambiguous shall be associated with the nearest possible \texttt{if} that would otherwise have no corresponding \texttt{else}.

**Semantics**

The production \texttt{IfStatement : if ( Expression ) Statement else Statement} is evaluated as follows:

1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is \texttt{false}, go to step 8.
5. Evaluate the first \texttt{Statement}.
6. If Result(5).\texttt{type} = \texttt{break} and Result(5).\texttt{target} occurs in the current label set, return (\texttt{normal, Result(5).value, empty}).
7. Return Result(5).
8. If Result(8).\texttt{type} = \texttt{break} and Result(8).\texttt{target} occurs in the current label set, return (\texttt{normal, Result(8).value, empty}).
9. Evaluate the second \texttt{Statement}.
10. If Result(9).\texttt{type} = \texttt{break} and Result(9).\texttt{target} occurs in the current label set, return (\texttt{normal, Result(9).value, empty}).
10. Return Result(8).

The production \textit{IfStatement : if ( Expression ) Statement} is evaluated as follows:

1. Evaluate \textit{Expression}.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is false, return (normal, empty, empty).
5. Evaluate \textit{Statement}.
6. If Result(5).\textit{type} = break and Result(5).\textit{target} occurs in the current label set, return (normal, Result(5).\textit{value}, empty).

7-6. Return Result(5).

12.6 Iteration statements

Syntax

\textit{IterationStatement}:

\begin{align*}
\text{do & Statement while ( Expression )}; \\
\text{while ( Expression ) Statement} \\
\text{for ( Expression opt ; Expression opt ; Expression opt ) Statement} \\
\text{for ( var VariableDeclarationList ; Expression opt ; Expression opt ) Statement} \\
\text{for ( LeftHandSideExpression in Expression ) Statement} \\
\text{for ( var Identifier Initializer opt in Expression ) Statement}
\end{align*}

12.6.1 The do...while Statement

The production \textit{do Statement while ( Expression )}; is evaluated as follows:

1. Let \textit{V} = empty.
2. Evaluate \textit{Statement}.
3. If Result(2).\textit{value} is not empty, let \textit{V} = Result(2).\textit{value}.
4. If Result(2).\textit{type} = continue and Result(2).\textit{target} is in the current label set, go to 2.
5. If Result(2).\textit{type} = break and Result(2).\textit{target} is in the current label set, return (normal, V, empty).
6. If Result(2) is an abrupt completion, return Result(2).
7. Evaluate Expression.
8. Call GetValue(Result(7)).
9. Call ToBoolean(Result(8)).
10. If Result(9) is true, go to step 2.
11. Return (normal, V, empty);

12.6.2 The while statement

The production \textit{IterationStatement : while ( Expression ) Statement} is evaluated as follows:

1. Let \textit{V} = empty.
2. Evaluate \textit{Expression}.
3. Call GetValue(Result(2)).
4. Call ToBoolean(Result(3)).
5. If Result(4) is false, return (normal, V, empty).
6. Evaluate \textit{Statement}.
7. If Result(6).\textit{value} is not empty, let \textit{V} = Result(6).\textit{value}.
8. If Result(6).\textit{type} = continue and Result(6).\textit{target} is in the current label set, go to 2.
9. If Result(6).\textit{type} = break and Result(6).\textit{target} is in the current label set, return (normal, V, empty).
10. If Result(6) is an abrupt completion, return Result(6).
11. Go to step 2.

12.6.3 The for statement

The production \textit{IterationStatement : for ( Expression opt ; Expression opt ; Expression opt ) Statement} is evaluated as follows:
1. If the first Expression is not present, go to step 4.
2. Evaluate the first Expression.
3. Call GetValue(Result(2)). (This value is not used.)
4. Let \( V = \text{empty} \).
5. If the second Expression is not present, go to step 10.
6. Evaluate the second Expression.
7. Call GetValue(Result(6)).
8. Call ToBoolean(Result(7)).
9. If Result(8) is false, go to step 19.
10. Evaluate Statement.
11. If Result(10).value is not empty, let \( V = \text{Result(10).value} \)
12. If Result(10).type = break and Result(10).target is in the current label set, go to step 19.
13. If Result(10).type = continue and Result(10).target is in the current label set, go to step 15.
14. If Result(10) is an abrupt completion, return Result(10).
15. If the third Expression is not present, go to step 5.
16. Evaluate the third Expression.
17. Call GetValue(Result(16)). (This value is not used.)
18. Go to step 5.

The production IterationStatement : for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement is evaluated as follows:

1. Evaluate VariableDeclarationList.
2. Let \( V = \text{empty} \).
3. If the second Expression is not present, go to step 8.
4. Evaluate the second Expression.
5. Call GetValue(Result(4)).
6. Call ToBoolean(Result(5)).
7. If Result(6) is false, go to step 15.
8. Evaluate Statement.
9. If Result(8).value is not empty, let \( V = \text{Result(8).value} \).
10. If Result(8).type = break and Result(8).target is in the current label set, go to step 17.
11. If Result(8).type = continue and Result(8).target is in the current label set, go to step 13.
12. If Result(8) is an abrupt completion, return Result(8).
13. If the third Expression is not present, go to step 3.
14. Evaluate the third Expression.
15. Call GetValue(Result(14)). (This value is not used.)
16. Go to step 3.
17. Return (normal, V, empty).

12.6.4 The for..in statement

The production IterationStatement : for ( LeftHandSideExpression in Expression ) Statement is evaluated as follows:

1. Evaluate the Expression.
2. Call GetValue(Result(1)).
3. Call ToObject(Result(2)).
4. Let \( V = \text{empty} \).
5. Get the name of the next property of Result(3) that doesn’t have the DontEnum attribute. If there is no such property, go to step 14.
6. Evaluate the LeftHandSideExpression (it may be evaluated repeatedly).
7. Call PutValue(Result(6), Result(5)).
8. Evaluate Statement.
9. If Result(8).value is not empty, let \( V = \text{Result(8).value} \).
10. If Result(8).type = break and Result(8).target is in the current label set, go to step 14.
11. If Result(8).type = continue and Result(8).target is in the current label set, go to step 5.
12. If Result(8) is an abrupt completion, return Result(8).

The production *IterationStatement : for (var VariableDeclaration in Expression ) Statement* is evaluated as follows:

1. Evaluate *VariableDeclaration*.
2. Evaluate *Expression*.
3. Call GetValue(Result(2)).
4. Call ToObject(Result(3)).
5. Let $V = \text{empty}$.
6. Get the name of the next property of Result(4) that doesn’t have the DontEnum attribute. If there is no such property, go to step 19.
7. Evaluate Result(1) as if it were an *Identifier*; see Error! Reference source not found.Error! Reference source not found.10.1.4 (yes, it may be evaluated repeatedly).
8. Call PutValue(Result(7), Result(6)).
9. Evaluate *Statement*.
10. If Result(9).value is not *empty*, let $V = \text{Result(9).value}$.
11. If Result(9).type = *break* and Result(9).target is in the current label set, go to step 15.
12. If Result(9).type = *continue* and Result(9).target is in the current label set, go to step 6.
13. If Result(8) is an abrupt completion, return Result(8).
15. Return (normal, V, empty).

The mechanics of enumerating the properties (step 5 in the first algorithm, step 6 in the second) is implementation dependent. The order of enumeration is defined by the object. Properties of the object being enumerated may be deleted during enumeration. If a property that has not yet been visited during enumeration is deleted, then it will not be visited. If new properties are added to the object being enumerated during enumeration, the newly added properties are not guaranteed to be visited in the active enumeration.

Enumerating the properties of an object includes enumerating properties of its prototype, and the prototype of the prototype, and so on, recursively; but a property of a prototype is not enumerated if it is “shadowed” because some previous object in the prototype chain has a property with the same name.

12.7 The *continue* statement

Syntax

*ContinueStatement :*

```
  continue [no LineTerminator here] Identifier_opt ;
```

Semantics

A program is considered syntactically incorrect if either of the following are true:

- The program contains a *continue* statement without the optional *Identifier*, which is not nested, directly or indirectly (but not crossing function boundaries), within an *IterationStatement*.
- The program contains a *continue* statement with the optional *Identifier*, where *Identifier* does not appear in the label set of an enclosing *IterationStatement*.

A *ContinueStatement* without an *Identifier* is evaluated as follows:

1. Return (continue, empty, empty).

A *continue* statement with the optional *Identifier* is evaluated as follows:

1. Return (continue, empty, *Identifier*).
12.8 **The break statement**

**Syntax**

```
BreakStatement :
    break [no LineTerminator here] Identifier opt ;
```

**Semantics**

A program is considered syntactically incorrect if either of the following are true:

- The program contains a `break` statement without the optional `Identifier`, which is not nested, directly or indirectly (but not crossing function boundaries), within an `IterationStatement` or a `SwitchStatement`.
- The program contains a `break` statement with the optional `Identifier`, where `Identifier` does not appear in the label set of an enclosing `Statement`.

A `BreakStatement` without an `Identifier` is evaluated as follows:

1. Return `(break, empty, empty)`. A `break` statement with an `Identifier` is evaluated as follows:

1. Return `(break, empty, Identifier)`.

12.9 **The return statement**

**Syntax**

```
ReturnStatement :
    return [no LineTerminator here] Expression opt ;
```

**Semantics**

An ECMAScript program is considered syntactically incorrect if it contains a `return` statement that is not within the `Block` of a `FunctionDeclaration`. It causes a function to cease execution and return a value to the caller. If `Expression` is omitted, the return value is the `undefined` value. Otherwise, the return value is the value of `Expression`.

The production `ReturnStatement :: return [no LineTerminator here] Expression opt ;` is evaluated as:

1. If the `Expression` is not present, return `(return, undefined, empty)`.  
2. Evaluate `Expression`.  
3. Call `GetValue(Result(2))`.  
4. Return `(return, Result(3), empty)`.

12.10 **The with statement**

**Syntax**

```
WithStatement :
    with ( Expression ) Statement
```

**Description**

The `with` statement adds a computed object to the front of the scope chain of the current execution context, then executes a statement with this augmented scope chain, then restores the scope chain.

**Semantics**

The production `WithStatement :: with ( Expression ) Statement` is evaluated as follows:

1. Evaluate `Expression`.  
2. Call `GetValue(Result(1))`.  
3. Call `ToObject(Result(2))`.  
4. Add `Result(3)` to the front of the scope chain.
5. Evaluate Statement using the augmented scope chain from step 4.
6. Remove Result(3) from the front of the scope chain.
7. Return Result(5).

Discussion
Note that no matter how control leaves the embedded Statement, whether normally or by some form of abrupt completion, the start of the scope chain is always restored to its former state.

12.11 The switch Statement

Syntax

SwitchStatement:
switch (Expression) CaseBlock

CaseBlock:
{ CaseClauses<opt> }
{ CaseClauses<opt> DefaultClause CaseClauses<opt> }

CaseClauses:
CaseClause
CaseClauses CaseClause

CaseClause:
case Expression : StatementList<opt>

DefaultClause:
default : StatementList<opt>

Semantics

The production SwitchStatement: switch (Expression) CaseBlock is evaluated as follows:
1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Evaluate CaseBlock, passing it Result(2) as a parameter.
4. If Result(3).type = break and Result(3).target is in the current label set, return (normal, Result(3).value, empty).
5. Return Result(3).

The production CaseBlock: { CaseClauses DefaultClause CaseClauses } is given an input parameter, input, and is evaluated as follows:
1. Let A be the list of CaseClause items in the first CaseClauses, in source text order.
2. For the next CaseClause in A, evaluate CaseClause. If there is no such CaseClause, go to step 7.
3. If input is not equal to Result(2), as defined by the !== operator, go to step 2.
4. Evaluate the StatementList of this CaseClause.
5. If Result(4) is an abrupt completion then return Result(4).
7. Let B be the list of CaseClause items in the second CaseClauses, in source text order.
8. For the next CaseClause in B, evaluate CaseClause. If there is no such CaseClause, go to step 15.
9. If input is not equal to Result(8), as defined by the !== operator, go to step 8.
10. Evaluate the StatementList of this CaseClause.
11. If Result(10) is an abrupt completion then return Result(10).
13. For the next CaseClause in A, evaluate the StatementList of this CaseClause. If there is no such CaseClause, go to step 15.
14. If Result(13) is an abrupt completion then return Result(13).
15. Execute the StatementList of DefaultClause.
16. If Result(15) is an abrupt completion then return Result(15).
17. Let $B$ be the list of $\text{CaseClause}$ items in the second $\text{CaseClauses}$, in source text order.
18. For the next $\text{CaseClause}$ in $B$, evaluate the $\text{StatementList}$ of this $\text{CaseClause}$. If there is no such $\text{CaseClause}$, return $(\text{normal, empty, empty})$.
19. If Result(18) is an abrupt completion then return Result(18).
20. Go to step 18.

The production $\text{CaseClause} : \text{case} \ \text{Expression} : \text{StatementList}_{opt}$ is evaluated as follows:

1. Evaluate $\text{Expression}$.
2. Call GetValue(Result(1)).
3. Return Result(2).

Note that evaluating $\text{CaseClause}$ does not execute the associated $\text{StatementList}$. It simply evaluates the $\text{Expression}$ and returns the value, which the $\text{CaseBlock}$ algorithm uses to determine which $\text{StatementList}$ to start executing.

12.12 Labeled Statements

Syntax

$LabeledStatement : Identifier : Statement$

Semantics

A $\text{Statement}$ may be prefixed by a label. Labeled statements are only used in conjunction with labeled $\text{break}$ and $\text{continue}$ statements. ECMAScript has no $\text{goto}$ statement.

An ECMAScript program is considered syntactically incorrect if it contains a $\text{LabeledStatement}$ that is enclosed by a $\text{LabeledStatement}$ with the same $\text{Identifier}$ as label. This does not apply to labels appearing within the body of a $\text{FunctionDeclaration}$ that is nested, directly or indirectly, within a labeled statement.

The production $\text{Identifier} : \text{Statement}$ is evaluated by adding $\text{Identifier}$ to the label set of $\text{Statement}$ and then evaluating $\text{Statement}$. If the $\text{LabeledStatement}$ itself has a non-empty label set, these labels are also added to the label set of $\text{Statement}$ before evaluating it. If the result of evaluating $\text{Statement}$ is $(\text{break, V, L})$ where $L$ is equal to $\text{Identifier}$, the production results in $(\text{normal, V, empty})$.

Prior to the evaluation of a $\text{LabeledStatement}$, the contained $\text{Statement}$ is regarded as possessing an empty label set, except if it is an $\text{IterationStatement}$ or a $\text{SwitchStatement}$, in which case it is regarded as possessing a label set consisting of the single element, $\text{empty}$.

12.10 The \texttt{throw} statement

Syntax

$\text{ThrowStatement} : \text{throw} \ \text{Expression} ;$

Semantics

The production $\text{ThrowStatement} :: \text{throw} \ \text{Expression} ;$ is evaluated as:

1. Evaluate $\text{Expression}$.
2. Call GetValue(Result(1)).
3. Return $(\text{throw, Result(2), empty})$, behaving as if a runtime error has occurred. See section 5.2.

12.11 The \texttt{try} statement

Syntax

$\text{TryStatement} : \text{try} \ \text{Block} \ \text{catch} \ (\text{var} \ \text{Identifier}) \ \text{Block}$

Description
The `try` statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a `throw` statement. The `catch` clause provides the exception handling code. The identifier introduces a local variable that is created when the execution scope containing the `try` statement is entered.

**Semantics**

The production `TryStatement :: try Block catch (var Identifier ...) Block ;` is evaluated as follows:

1. Evaluate the first `Block`.
2. If `Result(1).type` is not `throw`, return `Result(1)`.
3. Evaluate `Identifier`.
4. Call `PutValue(Result(3), V)`.
5. Evaluate the second `Block`.
6. Return `Result(5)`.

**Syntax**

```
TryStatement :
    try Block CatchList
    try Block Finally
    try Block CatchList Finally

CatchList :
    Catch
    CatchList Catch

Catch :
    catch (Identifier CatchGuard opt) Block

CatchGuard :
    : Expression

Finally :
    finally Block
```

**Description**

The `try` statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a `throw` statement. The `catch` clauses provide the exception-handling code. Entering a catch clause is similar to calling a function: there is a new execution context and the binding of a value to a formal parameter. The `finally` clause is executed just before control `finally` leaves a try block (that is, after any exception-handling code has been executed).

**Semantics**

The production `TryStatement : try Block CatchList` is evaluated as follows:

1. Evaluate `Block`.
2. If `Result(1).type` is not `throw`, return `Result(1)`,
3. Evaluate `CatchList` with parameter `Result(1)`,
4. If `Result(3) = (throw, empty, empty)`, return `Result(1)`
5. Return `Result(3)`.

The production `TryStatement : try Block Finally` is evaluated as follows:

1. Evaluate `Block`.
2. Evaluate `Finally`.
3. If `Result(2).type` is `normal`, return `Result(1)`.
4. Return `Result(2)`.

The production `TryStatement : try Block CatchList Finally` is evaluated as follows:
1. Evaluate `Block`.
2. Let `C = Result(1)`.
3. If `Result(1).type` is not `throw`, go to step 6.
4. Evaluate `CatchList` with parameter `Result(1)`.
5. Let `C = Result(4)`.
6. If `Result(4) = (throw, empty, empty)`, let `C = Result(1)`.
7. Evaluate `Finally`.
8. If `Result(7).type` is `normal`, return `C`.
9. Return `Result(7)`.

The production `CatchList : Catch` is evaluated as follows:
1. Evaluate `Catch` passing it the parameter passed to this production.
2. Return `Result(1)`.

The production `CatchList : CatchList Catch` is evaluated as follows:
1. Evaluate `CatchList` passing it the parameter passed to this production.
2. If `Result(1)` is not `throw, empty, empty)`, return `Result(1)`.
3. Evaluate `Catch` passing it the parameter passed to this production.
4. Return `Result(2)`.

The production `Catch : catch (Identifier CatchGuardopt) Block` is evaluated as follows:
1. Let `C = (throw, empty, empty)`.
2. Create a new `Object` object.
3. Call the `[[Put]]` method of `Result(2)` with parameters `Identifier` and `C.value`.
4. Add `Result(2)` to the front of the scope chain.
5. If there is no `CatchGuard`, go to step 10.
6. Evaluate `CatchGuard`.
7. If an exception value was thrown during the evaluation of `CatchGuard`, go to step 13.
8. If a runtime error occurred during the evaluation of `CatchGuard`, go to step 15.
9. If `ToBoolean(Result(6))` is not `true`, go to step 17.
10. Evaluate `Block`.
11. Let `C = Result(10)`.
12. Go to step 17.
13. Let `C = (throw, W, empty)` where `W` is the exception value thrown during the evaluation of `CatchGuard`.
14. Go to step 17.
15. Construct an appropriate `Error` object.
16. Let `C = (throw, Result(15), empty)`.
17. Remove `Result(2)` from the front of the scope chain.
18. Return `C`.

The production `CatchGuard : if Expression` is evaluated as follows:
1. Evaluate `Expression`.
2. Return `Result(1)`.

The production `Finally : finally Block` is evaluated as follows:
1. Evaluate `Finally`.
2. Return `Result(1)`.

14 Program

Syntax

Program : SourceElements
SourceElements :
  SourceElement
SourceElements SourceElement

SourceElement :
  Statement
  FunctionDeclaration

The production Program : SourceElements is evaluated as follows:
1. Process SourceElements for function declarations.
2. Evaluate SourceElements.
3. Return Result(2).

The production SourceElements : SourceElement is processed for function declarations as follows:
1. Process SourceElement for function declarations.

The production SourceElements : SourceElement is evaluated as follows:
1. Evaluate SourceElement.
2. Return Result(1).

The production SourceElements : SourceElements SourceElement is processed for function declarations as follows:
1. Process SourceElements for function declarations.

The production SourceElements : SourceElements SourceElement is evaluated as follows:
1. Evaluate SourceElements.
2. If Result(1) is an abrupt completion, return Result(1)
3-3. Evaluate SourceElement.
3-4. If Result(23).value = empty, let Result(23).value = Result(1).value.
4-5. Return Result(23).

The production SourceElement : Statement is processed for function declarations by taking no action.

The production SourceElement : Statement is evaluated as follows:
1. Evaluate Statement.
2. Return Result(1).

The production SourceElement : FunctionDeclaration is processed for function declarations as follows:
1. Process FunctionDeclaration for function declarations.

The production SourceElement : FunctionDeclaration is evaluated as follows:
1. Return (normal, empty, empty).

15.1.2 Function properties of the global object

15.1.2.1 eval(x)
When the eval function is called with one argument x, the following steps are taken:
1. If x is not a string value, return x.
2. Parse x as an ECMAScript Program. If the parse fails, generate a runtime error.
3. Evaluate the program from step 2.
4. If Result(3).type = throw, return Result(3), behaving as if a runtime error has occurred, see section 5.2.
5. If Result(3).value is not empty, return Result(3).value.
6. Return undefined.