Async Await Draft Spec

9. Functions

Functions abstract over executable actions.

functionSignature:
  metadata returnType? identifier formalParameterList
  ;

returnType:
  void
  | *type*
  ;

functionBody:
  ***async**? '=>' expression ';'*
  | *(**async** | **async*** | **sync***)? block*
  ;

block:
  '{' statements '}'
  ;

Functions include function declarations, methods, getters, setters and function literals.

All functions have a signature and a body. The signature describes the formal parameters of the function, and possibly its name and return type.
A function body is either:

- A block statement containing the statements executed by the function, optionally marked with one of the modifiers: **async**, **async*** or **sync***. In this case, if the last statement of a function is not a return statement, the statement **return**; is implicitly appended to the function body.

*Because Dart is optionally typed, we cannot guarantee that a function that does not return a value will not be used in the context of an expression. Therefore, every function must return a value. A **return** without an expression returns **null**. For generator functions, the situation is more subtle. See the discussion around the return statement.* OR

- of the form => e which is equivalent to a body of the form {**return** e;} or the form **async** => e which is equivalent to a body of the form **async** {**return** e;}. *The other modifiers do not apply here, because generators do not allow the form **return** e; values are added to the generated stream or iterable using **yield** instead.*

A function is *asynchronous* if its body is marked with the **async** or **async*** modifier. Otherwise the function is *synchronous*. A function is a *generator* if its body is marked with the **sync*** or **async*** modifier. It is a compile-time error if an **async**, **async*** or **sync*** modifier is attached to the body of a constructor.

Whether a function is synchronous or asynchronous is orthogonal to whether it is a generator or not. A generator function is a sugar for functions that produce collections in a systematic way, by lazily applying the generator function. Dart provides such a sugar in both the synchronous case, where one returns an iterable, and in the asynchronous case, where one returns a stream. Dart also allows both synchronous and asynchronous functions that produce a single value.

*One could allow modifiers for factories. A factory for Future could be modified by **async**, a factory for Stream could be modified by **async*** and a factory for Iterable could be modified by **sync***. No other scenario makes sense because the object returned by the factory would be of the wrong type. This situation is very unusual so it is not worth making an exception to the general rule for constructors in order to allow it.*

15.9 Throw

The *throw expression* is used to raise an exception.

throwExpression:
  throw *expression*
 ;

throwExpressionWithoutCascade:
  throw *expressionWithoutCascade*
 ;

The *current exception* is the last unhandled exception thrown.

Evaluation of a throw statement of the form **throw** *e* ; proceeds as follows:

The expression *e* is evaluated yielding a value *v.* If *e* evaluates to **null**, then a NullThrownError is thrown. Otherwise the current exception is set to *v* and the current return value becomes undefined.

The current exception and the current return value must never be simultaneously defined, as they represent mutually exclusive options for exiting the current function.

Let *f* be the immediately enclosing function.

If *f* is marked **async** or **async*** and there is a dynamically enclosing exception handler *h* introduced by the current activation, control is transferred to *h*, otherwise *f* terminates.

The rules for where a thrown exception will be handled must necessarily differ between the synchronous and asynchronous cases. Asynchronous functions cannot transfer control to an exception handler defined outside themselves. Asynchronous generators post exceptions to their stream. Other asynchronous functions report exceptions via their future.

Otherwise, control is transferred to the nearest dynamically enclosing exception handler.

If *f* is marked **sync*** then the dynamically enclosing exception handler encloses the call to moveNext() that initiated the evaluation of the throw expression.

There is no requirement that the expression *e* evaluate to a special kind of exception or error object.

If the object being thrown is an instance of class Error or a subclass thereof, its stackTrace getter will return the stack trace current at the point where the the object was first thrown.

The static type of a throw expression is bottom.

15.15 Function Invocation

Function invocation occurs in the following cases: when a function expression is invoked, when a method, getter or setter is invoked or when a constructor is invoked (either via instance creation , constructor redirection or super initialization). The various kinds of function invocation differ as to how the function to be invoked, *f*, is determined as well as whether **this** is bound. Once *f* has been determined, the formal parameters of *f* are bound to the corresponding actual arguments. The body of *f* is then executed with the aforementioned bindings.

If *f* is marked **async**, then an object *o* implementing class Future is associated with the invocation and immediately returned to the caller. The body of *f* is scheduled for execution at some future time. The future *o* will complete when *f* terminates. The value used to complete *o* is the current return value, if it is defined, and the current exception otherwise.

If *f* is marked **async***, then an object *s* implementing class Stream is associated with the invocation and immediately returned. When *s* is listened to, execution of the body of *f* will begin. When *f* terminates:

- If the current return value is defined then, if *s* has been canceled then its cancellation future is completed with **null**.
- If the current exception x is defined:
    - *x* is added to *s*.
    - If *s* has been canceled then its cancellation future is completed with *x*.
- *s* is closed.

When an asynchronous generator's stream has been canceled, cleanup will occur in the ***finally*** *clauses inside the generator. We choose to direct any exceptions that occur at this time to the cancellation future rather than have them be lost.*

If *f* is asynchronous then when *f* terminates any open stream subscriptions associated with any asynchronous for loops or **yield*** statements executing within *f* are canceled.

Such streams may be left open by for loops that were escaped when an exception was thrown within them for example.

If *f* is marked **sync***, then an object *i* implementing class Iterable is associated with the invocation and immediately returned. When iteration over the iterable is started, by getting an iterator *j* from the iterable and calling moveNext() on it, execution of the body of *f* will begin. When *f* terminates, *j* is positioned after its last element, so that its current value is **null** and the current call to moveNext() on *j* returns false, as will all further calls.

If *f* is synchronous and is not a generator then when *f* terminates the current return value is returned to the caller.

Execution of *f* terminates when the first of the following occurs:

- An uncaught exception is thrown
- A return statement nested in the body of *f* is executed and not intercepted in a **finally** clause.
- The last statement of the body completes execution.

15.xx Await Expressions

An *await expression* allows code to yield control until an asynchronous operation completes.

awaitExpression:
    await *expression;*

Evaluation of an await expression *a* of the form ***await*** *e* proceeds as follows:
First, the expression *e* is evaluated. Next:

- If *e* evaluates to an instance of Future, *f*, then execution of the function *m* immediately enclosing *a* is suspended until after *f* completes. The stream associated with the innermost enclosing asynchronous for loop, if any, is paused. At some time after *f* is completed, control returns to the current invocation. The stream associated with the innermost enclosing asynchronous for loop, if any, is resumed. If *f* has completed with an exception *x*, *a* raises *x*. If *f* completes with a value *v*, *a* evaluates to *v*.

- Otherwise, the value of *a* is the value of *e*. If evaluation of *e* raises an exception *x*, *a* raises *x*.

It is a compile-time error if the function immediately enclosing *a* is not declared asynchronous.

*An **await** expression has no meaning in a synchronous function. If such a function were to suspend waiting for a future, it would no longer be synchronous.*

It is not a static warning if the type of *e* is not a subtype of Future. Tools may choose to give a hint in such cases.

Let *flatten(T) = flatten(S) if T = Future<S>, and T otherwise.* The static type of *a* is *flatten(T)* where *T* is the static type of *e*.

We collapse multiple layers of futures into one. If e evaluates to a future f, the future will not invoke its then() callback until f completes to a non-future value, and so the result of an await or of an async function will never have type Future<X> where X itself is an invocation of Future.

15.31 Identifier Reference

An *identifier expression* consists of a single identifier; it provides access to an object via an unqualified name.

identifier:
    IDENTIFIER
  ;

IDENTIFIER_NO_DOLLAR:

*IDENTIFIER_START_NO_DOLLAR IDENTIFIER_PART_NO_DOLLAR**
;

IDENTIFIER:
*IDENTIFIER_START IDENTIFIER_PART**
;

;

BUILT_IN_IDENTIFIER:
abstract
| **as**
| **deferred**
| **dynamic**
| **export**
| **external**
| **factory**
| **get**
| **implements**
| **import**
| **library**
| **operator**
| **part**
| **set**
| **static**
| **typedef**
;

IDENTIFIER_START:
IDENTIFIER_START_NO_DOLLAR
| *'$'*
;

IDENTIFIER_START_NO_DOLLAR:
LETTER
| *'_'*
;

IDENTIFIER_PART_NO_DOLLAR:
IDENTIFIER_START_NO_DOLLAR
| *DIGIT*
;

IDENTIFIER_PART:

```
    IDENTIFIER_START
  | DIGIT
  ;
```

qualified:

```
    identifier ('.' identifier)?
  ;
```

A built-in identifier is one of the identifiers produced by the production BUILT_IN_IDENTIFIER. It is a compile-time error if a built-in identifier is used as the declared name of a class, type parameter or type alias. It is a compile-time error to use a built-in identifier other than **dynamic** as a type annotation.

Built-in identifiers are identifiers that are used as keywords in Dart, but are not reserved words in Javascript. To minimize incompatibilities when porting Javascript code to Dart, we do not make these into reserved words. A built-in identifier may not be used to name a class or type. In other words, they are treated as reserved words when used as types. This eliminates many confusing situations without causing compatibility problems. After all, a Javascript program has no type declarations or annotations so no clash can occur. Furthermore, types should begin with an uppercase letter (see the appendix) and so no clash should occur in any Dart user program anyway.

It is a compile-time error if any of the identifiers **async**, **await** or **yield** is used as an identifier in a function body marked with either **async**, **async*** or **sync***.

For compatibility reasons, new constructs cannot rely upon new reserved words or even built-in identifiers. However, the constructs above are only usable in contexts that require special markers introduced concurrently with these constructs, so no old code could use them. Hence the restriction, which treats these names as reserved words in a limited context.

Evaluation of an identifier expression *e* of the form *id* proceeds as follows:
Let *d* be the innermost declaration in the enclosing lexical scope whose name is *id* or *id=*. If no such declaration exists in the lexical scope, let *d* be the declaration of the inherited member named *id* if it exists.

- If *d* is a class or type alias *T*, the value of *e* is an instance of class Type reifying *T*.
- If *d* is a type parameter *T*, then the value of *e* is the value of the actual type argument

corresponding to *T* that was passed to the generative constructor that created the current binding of **this**. If, however, *e* occurs inside a static member, a compile-time error occurs.

- If *d* is of one of the forms ***const*** *v = e;* or ***const*** *T v = e;* the result of the getter is the value of the compile time constant *e*.
- If *d* is a local variable or formal parameter then *e* evaluates to the current binding of *id.*
- If *d* is a static method, top level function or local function then *e* evaluates to the function defined by *d.*
- If *d* is the declaration of a static variable, static getter or static setter declared in class *C,* then *e* is equivalent to the getter invocation *C.id.*
- If *d* is the declaration of a library variable, top-level getter or top level setter, then *e* is equivalent to the getter invocation *id.*
- Otherwise, if *e* occurs inside a top level or static function (be it function, method, getter, or setter) or variable initializer, evaluation of *e* causes a NoSuchMethodError to be thrown.
- Otherwise *e* is equivalent to the property extraction **this**.*id.*

The static type of *e* is determined as follows:

- If *d* is a class, type alias or type parameter the static type of *e* is Type.
- If *d* is a local variable or formal parameter the static type of *e* is the type of the variable *id*, unless *id* is known to have some type *T* in which case the static type of *e* is *T* , provided that *T* is a more specific than any other type *S* such that v is known to have type *S*.
- If *d* is a static method, top-level function or local function the static type of *e* is the function type defined by *d*.
- If *d* is the declaration of a static variable, static getter or static setter declared in class *C*, the static type of *e* is the static type of the getter invocation *C.id*.
- If *d* is the declaration of a library variable, top-level getter or top-level setter, the static type of *e* is the static type of the getter invocation *id*.
- Otherwise, if *e* occurs inside a top level or static function (be it function, method, getter, or setter) or variable initializer, the static type of *e* is **dynamic**.
- Otherwise, the static type of *e* is the type of the property extraction **this**.*id.*

Note that if one declares a setter, we bind to the corresponding getter even if it does not exist.

This prevents situations where one uses uncorrelated setters and getters. The intent is to prevent errors when a getter in a surrounding scope is used accidentally.

It is as static warning if an identifier expression of the form *id* occurs inside a top level or static function (be it function, method, getter, or setter) or variable initializer and there is no declaration *d* with name *id* in the lexical scope enclosing the expression.

16.6 For

The *for statement* supports iteration.

forStatement:
    await*?* **for** *'(' forLoopParts ')' statement*
  ;

forLoopParts:
    forInitializerStatement expression? ';' expressionList?
  | *declaredIdentifier* **in** *expression*
  | *identifier* **in** *expression*
  ;

forInitializerStatement:
    localVariableDeclaration ';'
  | *expression? ';'*
  ;

The for statement has two forms - the traditional for loop and the foreach statement.

16.6.3 Asynchronous For-in

A foreach statement may be asynchronous. The asynchronous form is designed to iterate over streams. An asynchronous for loop is distinguished by the keyword **await** immediately preceding the keyword **for**.

Execution of a foreach statement of the form **await for** (*finalConstVarOrType?* id **in** *e*) *s* is executed as follows:

The expression *e* is evaluated to an object *o*. It is a dynamic error if *o* is not an instance of a class that implements Stream. Otherwise, a new future *f* is created, and the expression **await** *f* is evaluated.

The stream o is listened to, and on each data event in the stream *o* the statement *s* is executed with id bound to the value of the current element of the stream. If *s* raises an exception, or if *o* raises an exception, then *f* is completed with that exception. Otherwise, when all events in the stream *o* have been processed, *f* is completed with **null**.

If another event of *u* occurs before execution of *s* is complete, handling of it must wait until *s* is complete.

*The future f and the corresponding **await** expression ensure that execution suspends as an asynchronous for loop begins and resumes after the **for** statement when it ends. They also ensure that the stream of any enclosing asynchronous for loop is paused for the duration of this loop.*

It is a compile-time error if an asynchronous foreach statement appears inside a synchronous function. It is a compile-time error if a traditional for loop is prefixed by the **await** keyword.

Again, an asynchronous loop would make no sense within a synchronous function.

16.10 Rethrow

The *rethrow statement* is used to re-raise an exception.

rethrowStatement:
  rethrow
 ;

Execution of a **rethrow** statement proceeds as follows:

Let *f* be the immediately enclosing function, and let **on** *T* **catch** (p_1, p_2) be the immediately enclosing catch clause.

*A **rethrow** always appears inside a catch clause, and any catch clause is semantically equivalent to some catch clause of the form **on** T **catch** (p_1, p_2). So we can assume that the **rethrow** is enclosed in a catch clause of that form.*

The current exception is set to *p1* , the current return value becomes undefined, and the active stack trace is set to *p2*.

If *f* is marked **async** or **async*** and there is a dynamically enclosing exception handler *h* introduced by the current activation, control is transferred to *h*, otherwise *f* terminates.

In the case of an asynchronous function, the dynamically enclosing exception handler is only relevant within the function. If an exception is not caught within the function, the exception value is channelled through a future or stream rather than propagating via exception handlers.

Otherwise, control is transferred to the nearest dynamically enclosing exception handler.

The change in control may result in multiple functions terminating if these functions do not catch the exception via a **catch** or **finally** clause, both of which introduce a dynamically enclosing exception handler.

It is a compile-time error if a **rethrow** statement is not enclosed within an on-catch clause.

Try

The *try statement* supports the definition of exception handling code in a structured way.

tryStatement:
    try *block (onPart+ finallyPart? | finallyPart)*
  ;

onPart:
    catchPart block
  | **on** *type catchPart? block*

  ;

catchPart:
    catch *'(' identifier (',' identifier)? ')'*
  ;

finallyPart:
    finally *block*
  ;

A try statement consists of a block statement, followed by at least one of:

1. A set of **on-catch** clauses, each of which specifies (either explicitly or implicitly) the type of exception object to be handled, one or two exception parameters and a block

statement.
2. A **finally** clause, which consists of a block statement.

The syntax is designed to be upward compatible with existing Javascript programs. The ***on*** *clause can be omitted, leaving what looks like a Javascript catch clause.*

An **on-catch** clause of the form **on** *T* **catch** (p_1, p_2) *s* or **on** *T s matches* an object *o* if the type of *o* is a subtype of *T.* If *T* is a malformed type, then performing a match causes a runtime error.

It is of course a static warning if *T* is a malformed type.

An **on-catch** clause of the form **on** *T* **catch** (p_1, p_2) *s* introduces a new scope *CS* in which local variables specified by p_1 and p_2 are defined. The statement s is enclosed within *CS.*

An **on-catch** clause of the form **on** *T* **catch** (p_1) *s* is equivalent to an **on-catch** clause **on** *T* **catch** (p_1, p_2) *s,* where p_2 is an identifier that does not occur anywhere else in the program.

An **on-catch** clause of the form **catch** (*p*) *s* is equivalent to an an **on-catch** clause **on dynamic catch** (*p*) *s.* An **on-catch** clause of the form **catch** (p_1, p_2) *s* is equivalent to an an **on-catch** clause **on dynamic catch** (p_1, p_2) *s.*

The *active stack trace* is an object whose toString() method produces a string that is a record of exactly those function activations within the current isolate that had not completed execution at the point where the current exception was thrown.

This implies that no synthetic function activations may be added to the trace, nor may any source level activations be omitted. This means, for example, that any inlining of functions done as an optimization must not be visible in the trace. Similarly, any synthetic routines used by the implementation must not appear in the trace.

Nothing is said about how any native function calls may be represented in the trace.

For each such function activation, the active stack trace includes the name of the function, the bindings of all its formal parameters, local variables and this, and the position at which the function was executing.

The term position should not be interpreted as a line number, but rather as a precise position - the exact character index of the expression that raised the exception.

The definition below is an attempt to characterize exception handling without resorting to a normal/abrupt completion formulation. It has the advantage that one need not specify abrupt completion behavior for every compound statement. On the other hand, it is new different and *needs more thought.*

A try statement **try** *s_1 on-catch$_1$* **…** *on-catch$_n$* **finally** *s_f* defines an exception handler *h* that executes as follows:

The **on-catch** clauses are examined in order, starting with *on-catch$_1$*, until either an **on-catch** clause that matches the current exception is found, or the list of **on-catch** clauses has been exhausted. If an **on-catch** clause *on-catch$_k$* is found, it is executed. Afterwards, whether or not an **on-catch** clause is found, the **finally** clause is executed. Then, execution resumes at the end of the try statement.

A **finally** clause **finally** *s* defines an exception handler *h* that executes as follows:

Let *r* be the current return value. Then the current return value becomes undefined. Any open streams associated with any asynchronous for loops and **yield*** statements executing within the dynamic scope of *h* are canceled.

Streams left open by for loops that were escaped for whatever reason would be canceled at function termination, but it is best to cancel them as soon as possible.

Then the **finally** clause is executed. If *r* is defined then the current return value is set to *r* and then:

- if there is a dynamically enclosing error handler *h* defined by a **finally** clause in *m*, control is transferred to *h*.
- Otherwise *m* terminates.

Otherwise, execution resumes at the end of the try statement.

Execution of an **on-catch** clause **on** *T* **catch** (p_1, p_2) *s* of a **try** statement *t* proceeds as follows:

Variables p_1 of static type *T,* and p_2 of static type StackTrace are implicitly declared, with a scope comprising *s*. The variable p_1 is bound to the current exception, and p_2 is bound to the active stack trace. Then, the current exception and active stack trace both become undefined.

Next, the statement *s* is executed in the dynamic scope of the exception handler defined by the finally clause of *t*.

Execution of a finally clause **finally** *s* of a try statement proceeds as follows:

Let *x* be the current exception and let *t* be the active stack trace. Then the current exception and the active stack trace both become undefined. The statement *s* is executed. Then, if *x* is defined, it is rethrown as if by a **rethrow** statement enclosed in a catch clause of the form **catch** (x, t).

Execution of a try statement of the form **try** *s_1 on-catch$_1$* **...** *on-catch$_n$* **finally** *s_f* proceeds as follows:

The statement *s_1* is executed in the dynamic scope of the exception handler defined by the try statement. Then, the **finally** clause is executed.

Whether any of the on-catch clauses is executed depends on whether a matching exception has been raised by *s_1* (see the specification of the **throw** statement).

If *s_1* has raised an exception, it will transfer control to the try statement's handler, which will examine the **on-catch** clauses in order for a match as specified above. If no matches are found, the handler will execute the **finally** clause.

If a matching handler was found, it will execute first, and then the **finally** clause will be executed.

If an exception is thrown during execution of an **on-catch** clause, this will transfer control to the handler for the **finally** clause, causing the **finally** clause to execute in this case as well.

If no exception was raised, the **finally** clause is also executed. Execution of the **finally** clause could also raise an exception, which will cause transfer of control to the next enclosing handler.

A try statement of the form **try** *s_1 on-catch$_1$* **...** *on-catch$_n$* is equivalent to the statement **try** *s_1 on-catch$_1$* **...** *on-catch$_n$* **finally** *{}*

16.12 Return

The *return statement* returns a result to the caller of a synchronous function, and completes the future (respectively adds to the stream) associated with an asynchronous function.

returnStatement:
  return *expression? ';'*
  ;

Due to **finally** clauses, the precise behavior of **return** is a little more involved. Whether the value a return statement is supposed to return is actually returned depends on the behavior of any **finally** clauses in effect when executing the return. A **finally** clause may choose to return another value, or throw an exception, or even redirect control flow leading to other returns or throws. All a return statement really does is set a value that is to be returned when the function terminates.

For generators, the situation is a bit different. The **yield** statement will immediately add values to the output iterable or stream.

The *current return value* is a unique value specific to a given function activation. It is undefined unless explicitly set by a **return** statement.

Executing a return statement

return *e*;

proceeds as follows:

First the expression *e* is evaluated producing an object *o.*

Next:

- The current return value is set to *o* and the current exception and active stack trace become undefined*.*
- Let *c* be the **finally** clause of the innermost enclosing **try-finally** statement, if any. If *c* is defined, let *h* be the handler induced by *c*. If *h* is defined, control is transferred to *h*.
- Otherwise execution of the current method terminates.

The enclosing function cannot be marked a generator, since it would not be allowed to contain a statement of the form **return** e;wait.. see below.

It is a static type warning if the type of *e* may not be assigned to the declared return type of the immediately enclosing function.

In checked mode, it is a dynamic type error if *o* is not **null** and the runtime type of *o* is not a subtype of the actual return type of the immediately enclosing function.

It is a compile-time error if a return statement of the form **return** *e*; appears in a generative constructor or in a generator function.

It is quite easy to forget to add the factory prefix for a constructor, accidentally converting a *factory into a generative constructor. The static checker may detect a type mismatch in some, but not all, of these cases. The rule above helps catch such errors, which can otherwise be very hard to recognize. There is no real downside to it, as returning a value from a generative constructor is meaningless.*

*In the case of a generator function, the value returned by the function is the iterable or stream associated with it, and individual elements are added to that iterable using **yield** statements, and so returning a value makes no sense.*

Let *f* be the function immediately enclosing a return statement of the form **return**; It is a static warning if the return type of *f* may not be assigned to **void** and *f* is neither a generator nor a generative constructor.

Hence, a static warning will not be issued if *f* has no declared return type, since the return type would be **dynamic** and **dynamic** may be assigned to **void**. However, any non-generator function that declares a return type must return an expression explicitly.

This helps catch situations where users forget to return a value in a return statement.

A return statement with no expression, **return**; is executed as follows:

If the immediately enclosing function *f* is a generator, then:

- The current return value is set to **null**.
- Let *c* be the **finally** clause of the innermost enclosing **try-finally** statement, if any. If *c* is defined, let *h* be the handler induced by *c*. If *h* is defined, control is transferred to *h*.
- Otherwise, execution of the current method terminates.

Otherwise the return statement is executed by executing the statement **return null**; if it occurs inside a method, getter, setter or factory; otherwise, the return statement necessarily occurs inside a generative constructor, in which case it is executed by executing **return this**;.

Despite the fact that **return**; is executed as if by a **return** *e*;, it is important to understand that it is **not** a static warning to include a statement of the form **return**; in a generative constructor. The rules relate only to the specific syntactic form **return** *e*;**.**

*The motivation for formulating **return**; in this way stems from the basic requirement that all function invocations indeed return a value. Function invocations are expressions, and we cannot rely on a mandatory typechecker to always prohibit use of **void** functions in expressions. Hence, a return statement must always return a value, even if no expression is specified.*

The question then becomes, what value should a return statement return when no return expression is given. In a generative constructor, it is obviously the object being constructed *(**this**). In void functions we use **null**. A void function is not expected to participate in an expression, which is why it is marked **void** in the first place. Hence, this situation is a mistake which should be detected as soon as possible. The static rules help here, but if the code is executed, using **null** leads to fast failure, which is desirable in this case. The same rationale applies for function bodies that do not contain a return statement at all.*

It is a static warning if a function contains both one or more explicit return statements of the form **return**; and one or more return statements of the form **return** *e*; .

Let *T* be the static type of *e* and let *f* be the immediately enclosing function. It is a static type warning if the body of *f* is marked **async** and the type *Future<flatten(T)>* may not be assigned to the declared return type of *f*.

Let *S* be the runtime type of *o*. In checked mode, it is a dynamic type error if *o* is not **null** and *Future<S>* is not a subtype of the actual return type of the immediately enclosing function.

16.xx Yield

16.xx.1

The **yield** statement adds an element to the result of a generator function.

yieldStatement:
  yield *expression* *';'*
  ;

Execution of a statement *s* of the form **yield** *e*; proceeds as follows:

First, the expression *e* is evaluated to an object *o*. If the enclosing function *m* is marked **async*** and the stream *u* associated with *m* has been paused, then execution of *m* i is suspended until *u* is resumed or canceled.

Next, *o* is added to the iterable or stream associated with the immediately enclosing function.

If the enclosing function *m* is marked **async*** and the stream *u* associated with *m* has been canceled, then let *c* be the **finally** clause of the innermost enclosing **try-finally** statement, if any. If *c* is defined, let *h* be the handler induced by *c*. If *h* is defined, control is transferred to *h*. If *h* is undefined, the immediately enclosing function terminates.

The stream associated with an asynchronous generator could be canceled by any code with a reference to that stream at any point where the generator was passivated. Such a cancellation constitutes an irretrievable error for the generator. At this point, the only plausible action for the generator is to clean up after itself via its ***finally*** *clauses.*

If the enclosing function *m* is marked **sync*** then:

- Execution of the function *m* immediately enclosing *s* is suspended until the method moveNext() is invoked upon the iterator used initiate the current invocation of *m*.
- The current call to moveNext() returns true.

It is a compile-time error if a **yield** statement appears in a function that is not a generator function.

Let *T* be the static type of *e* and let *f* be the immediately enclosing function. It is a static type warning if either:

- the body of *f* is marked **async*** and the type *Stream<T>* may not be assigned to the declared return type of *f*.
- the body of *f* is marked **sync*** and the type *Iterable<T>* may not be assigned to the declared return type of *f*.

16.xx.2 Yield*

The **yield*** statement adds a series of values to the result of a generator function.

yieldEachStatement:
  yield* *expression* ';'
  ;

Execution of a statement *s* of the form **yield*** *e*; proceeds as follows:

First, the expression *e* is evaluated to an object *o*. If the immediately enclosing function *m* is synchronous, then it is a dynamic error if the class of *o* does not implement Iterable. If *m* asynchronous, then it is a dynamic error if the class of *o* does not implement Stream. Next, for each element *x* of *o*:

- If *m* is marked **async*** and the stream *u* associated with *m* has been paused, then execution of *m* i is suspended until *u* is resumed or canceled.
- *x* is added to the iterable or stream associated with *m* in the order it appears in *o*.

- If the *m* is marked **async*** and the stream *u* associated with *m* has been canceled, then let *c* be the **finally** clause of the innermost enclosing **try-finally** statement, if any. If *c* is defined, let *h* be the handler induced by *c*. If *h* is defined, control is transferred to *h*. If *h* is undefined, the immediately enclosing function terminates.

If the enclosing function is marked **sync*** then:

- Execution of the function *m* immediately enclosing *s* is suspended until the method moveNext() is invoked upon the iterator used initiate the current invocation of *m*.
- The current call to moveNext() returns true.

It is a compile-time error if a **yield*** statement appears in a function that is not a generator function.

Let *T* be the static type of *e* and let *f* be the immediately enclosing function. It is a static type warning if *T* may not be assigned to the declared return type of *f*.

16.14 Break

The *break statement* consists of the reserved word **break** and an optional label.

breakStatement:
  break *identifier? ';'*
 ;

Let s_b be a break statement. If s_b is of the form **break** *L*; then let s_E be the the innermost labeled statement with label *L* enclosing s_b. If s_b is of the form **break**; then let s_E be the the innermost **do**, **for**, **switch** or **while** statement enclosing s_b. It is a compile-time error if no such statement s_E exists within the innermost function in which s_b occurs. Furthermore, let $s_1 \ldots s_n$ be those try statements that are both enclosed in s_E and that enclose s_b, and that have a **finally** clause. Lastly, let f_j be the **finally** clause of s_j, $1 <= j <= n$. Executing s_b first executes $f_1 \ldots f_n$ in innermost-clause-first order and then terminates s_E. If s_E is an asynchronous for loop, its associated stream subscription is canceled. Furthermore, let a_k be the set of asynchronous for loops and **yield*** statements enclosing s_b that are enclosed in s_E, $1 <= k <= m$. The stream subscriptions associated with a_j are canceled, 1 <= k <= m.

16.15 Continue

The *continue statement* consists of the reserved word **continue** and an optional label.

continueStatement:
  continue *identifier? ';'*
    ;

Let s_c be a continue statement. If s_c is of the form **continue** *L*; then let s_E be the the innermost labeled **do**, **for** or **while** statement or case clause with label *L* that encloses s_c. If s_c is of the form **continue**; then let s_E be the the innermost **do**, **for** or **while** statement enclosing s_c. It is a compile-time error if no such statement or case clause s_E exists within the innermost function in which s_c occurs. Furthermore, let $s_1 \dots s_n$ be those try statements that are both enclosed in s_E and that enclose s_c, and that have a **finally** clause. Lastly, let f_j be the **finally** clause of s_j, $1 <= j <= n$. Executing s_c first executes $f_1 \dots f_n$ in innermost-clause-first order. Then, if s_E is a case clause control is transferred to the case clause; otherwise, s_E is necessarily a loop and execution resumes after the last statement in the loop body.

In a while loop, that would be the boolean expression before the body. In a do loop, it would be the boolean expression after the body. In a for loop, it would be the increment clause. In other words, execution continues to the next iteration of the loop.

If s_E is an asynchronous for loop, let a_k be the set of asynchronous for loops and **yield*** statements enclosing s_c that are enclosed in s_E, $1 <= k <= m$. The stream subscriptions associated with a_j are canceled, 1 <= k <= m.