

Principles of Programming Languages

Asynchronous Programming

In this chapter, we will investigate the general topic of how to manage control flow in programming languages. Control flow is the order in which operations are executed within a program. In procedural languages, the simplest form of control flow is the sequence (execute statements one after the other).

Conditional statements (**if**, **switch**) determine which of multiple continuations are followed depending on the value of a tested expression. Loops determine how often a block of operations are executed depending on the value of a tested expression.

In functional languages, the key control flow is *function invocation*: the body of the function is executed within a scope in which the arguments are bound to the parameters of the invocation, and then the flow returns to the context in which the invocation took place.

Types of Control Flows

In addition to these classical control flows, programming languages include *non local control flow*: in these constructs, flows exits from a context of execution and continues at a predefined point. These include exceptions (`try/catch/finally/throw`), coroutines, generators, `async/await`, and continuation.

The study of control flow also includes a description of concurrency such as threads and their synchronization.

Interpreters and Control Flow: Asynchronous Programming and Continuation Passing Style

In the interpreters we have designed in Chapter 2, we did not explicitly model the call stack. The reason is that we relied on the call stack mechanism of the meta-language to provide the proper control of function calls and the return to their calling location.

Interpreters and Control Flow: Asynchronous Programming and Continuation Passing Style

In this sense, the interpreter we wrote does not *explain* how control flow is implemented in the object language, because we use the control flow primitives of the meta language.

Interpreters and Control Flow: Asynchronous Programming and Continuation Passing Style

In this chapter, we investigate the mechanisms through which programming languages provide control flow, and variants of control flow across different programming paradigms.

Interpreters and Control Flow: Asynchronous Programming and Continuation Passing Style

In particular, we present practical issues in Asynchronous Programming, and techniques allowing proper management of asynchronous tasks: callbacks, promises, and co-routines.

Interpreters and Control Flow: Asynchronous Programming and Continuation Passing Style

We review these topics in TypeScript - where they practically fill an important role both in the domains of client user interface (to deal with User Interface events in a reactive manner) and of backend servers (to deal with protocol implementation which are IO-bound in a resource efficient manner).

Interpreters and Control Flow: Asynchronous Programming and Continuation Passing Style

We then introduce a technique called **continuation passing style (CPS)**, investigate its properties and relate it to the way our interpreters model control flow, recursion and iteration.

Interpreters and Control Flow: Asynchronous Programming and Continuation Passing Style

We switch back to Scheme and to our Interpreter models and demonstrate a systematic transformation from recursive to CPS style and introduce techniques which elucidate the way asynchronous and lazy programming techniques are designed.

Interpreters and Control Flow: Asynchronous Programming and Continuation Passing Style

Finally, we use the notion of *continuation* that we have made concrete in the CPS transformation, and use it to re-implement the interpreter of Chapter 2 with explicit description of control flow as general continuations. In this model, the interpreter is modeled as a function `eval(exp, env, cont) => (value, cont)` where `cont` represents the control state of the interpreter.

Interpreters and Control Flow: Asynchronous Programming and Continuation Passing Style

At each step of the computation as explained by the operational semantics, the interpreter computes which expression to evaluate next and where to pass the result of this evaluation.

We implement this new approach in the language *L7*.

When a function f calls a function g , g needs to know where to return to (inside f) after it is done. This information is usually managed with a stack, the call stack. Let's look at an example:

Function Invocation - Call Stack

```
function h(z) {  
  // Print stack trace  
  console.log(new Error().stack); // (A)  
}  
function g(y) {  
  h(y + 1); // (B)  
}  
function f(x) {  
  g(x + 1); // (C)  
}  
f(3); // (D)
```

Initially, when the program is started, the call stack is empty.

After the function call `f(3)` in line `D`, the stack
has one entry:

`Location in global scope`

After the function call `g(x + 1)` in line C, the stack has two entries:

Location in f
Location in global scope

After the function call `h(y + 1)` in line **B**, the stack has three entries:

Location in `g`

Location in `f`

Location in global scope

The stack trace printed in line **A** shows you what the call stack looks like:

Error

```
at h (...:3:15)
at g (...:6:3)
at f (...:9:3)
```

Next, each of the functions terminates and each time the top entry is removed from the stack. After function **f** is done, we are back in global scope and the call stack is empty.

The mechanism of a call stack to manage function calls and their return is present in all languages.

The call stack is managed implicitly at runtime, and is usually not exposed to the programmer.

In JavaScript, as demonstrated above, the **Error()** object provides access to the current state of the call stack - so that we can inspect it (this is mostly useful when debugging errors at runtime). A similar facility exists in Java with the method **Thread.currentThread().getStackTrace()**.

JavaScript as a programming language is used in two main contexts:

- Inside Web browsers to implement user interface client code
- Inside Node processes on the backend to perform protocol handling code - usually, exposing access to resources as a REST API.

In both of these application domains, it is useful to think of the task of the JavaScript interpreter as processing an **event loop**.

The following 25 minute presentation: [Philip Roberts: What the heck is the event loop anyway? | JSConf EU 2014](#) provides a useful explanation of what is the event loop in the context of Web browsers.

The companion tool to this presentation visualizes the mechanism of the event loop: <http://latentflip.com/loupe/>.

Consider first a browser context. The browser event loop executes browser-related tasks that are fed into the browser task queue.

Typical browser tasks are generated when an HTML page is loaded into a browser tab, and they include:

- Parsing HTML
- Executing JavaScript code in script elements
- Reacting to user input (mouse clicks, key presses, etc.)
- Processing the result of asynchronous network requests (calls to HTTP server)

The last 3 are tasks that run JavaScript code, in the JavaScript interpreter embedded in the browser. These tasks terminate when the code terminates. Then the next task from the queue can be executed.

For example, browsers offer a timer facility:
`setTimeout()` creates a timer, waits until it fires and then adds a task to the queue. It has the signature:

```
setTimeout(callback, ms)
```

After `ms` milliseconds, `callback` is added to the task queue. It is important to understand that `ms` only specifies when the callback is added, not when it actually is executed. That may happen much later, especially if the event loop is heavily loaded.

Practically, the `setTimeout` primitive is a way to submit a task to the event loop for later execution.

Event Loops in the Browser

```
setTimeout(() => console.log("Later..."), 1000);  
console.log("Immediately");
```

```
/*  
    Immediately  
    // after ~1 second  
    Later...  
*/
```

In the `setTimeout` example, the first parameter is a function passed as an argument (taking advantage of the support for closures in the JavaScript language). The task that is posted to the event queue is called a **callback** (because it is called back by the event loop when it can instead of being called by the programmer directly).

Note how the task is represented: it is a procedure of no parameters. What we submit to the task queue is a closure. The idiom: `() => { ... }` indicates we “package” code to be executed later. When and who executes it later can be decided by the programmer or by the runtime environment.

Event Loops in the Browser

Another typical example of callbacks in the browser context is to associate callbacks to User Interface widgets. A typical example would be:

```
$("#btn_1").click(() => alert("Btn 1 Clicked"));
```

The anonymous function passed as an argument to the `click()` method defines a callback which is invoked by the Browser event loop whenever the button is clicked.

In the Node.js context, the event loop is driven by events related to asynchronous I/O calls. For example, when invoking the file system, to open a file, the time taken by the operating system (OS) system call is extremely large compared to the time it takes to execute a function call.

Instead of making the interpreter block and wait for this system call to complete, the Node interpreter submits a task to the event loop, which consists of invoking the callback of the system call when it has completed.

This strategy requires the programmer to change fundamentally the way I/O calls are organized. All the primitive calls to the File System (**fs**) module take a function as an argument - which is called the **callback** which is invoked when the slow FS operation has completed.

Compare the synchronous and asynchronous versions of a file system call:

Event Loops in Node

```
// Synchronous (blocking) call to readFileSync  
// The return value of the readFileSync procedure  
// can be passed directly to the JSON.parse function.  
const readJSONSync = (filename) => {  
  return JSON.parse(fs.readFileSync(filename, 'utf8'));  
}  
  
const writeJSONSync = (filename, map) => {  
  return fs.writeFileSync(filename,  
    JSON.stringify(map),  
    'utf8');  
}  
  
writeJSONSync("test", {id:1, text:'hello'});  
console.log(readJSONSync("test"));
```

Event Loops in Node

```
// Asynchronous version using callbacks
const fs = require('fs');

const readJSON = (filename, callback) => {
  fs.readFile(filename, 'utf8', (err, res) => {
    if (err) callback(err, undefined);
    else callback(undefined, JSON.parse(res));
  });
}

const writeJSON = (filename, map) => {
  fs.writeFile(filename, JSON.stringify(map), (err) => {
    if (err) console.error(err);
    else console.log("The file was saved: ", filename);
  });
  console.log("This is invoked before the callback is invoked.");
}

writeJSON("test.async", {id:1, test:'async'});
readJSON("test.async", (err, res) => { console.log(res); });
```

In the example above, the `fs.writeFile` Node method takes 3 arguments:

- The name of the file to create
- The string to write into the file
- The callback to invoke when the `writeFile` operation has completed.

This pattern takes advantage of the fact that JavaScript allows passing functions as arguments to other functions: the callback is an argument passed to the file system primitive.

Programming with callbacks is not easy. We cannot make the usual time-sequencing assumptions we make in sequential code. In the example above, we know that the expression immediately after the call to `fs.writeFile()` is executed before the callback is invoked. This guarantee is part of the semantics of the Node interpreter.

But we cannot make any assumption as to **when** the callback will be executed. It can take a very long time. The problem for the programmer is to decide where and how to write code which **depends** on the completion of the file system operation. Answering this question will be the main topic of this chapter.

Thinking of the JavaScript engine as a **reactive event loop** instead of an active process which invokes procedures is an important change of perspective - which is related to the design pattern of **inversion of control** and which defines the **event-driven** programming paradigm.

In asynchronous programming, callbacks are not invoked in the same call stack as the procedure that creates the callback. Instead, the function which generates the task (the asynchronous call) executes, and when it ends, it creates a task (a closure) which is added to the task queue.

When the event loop determines that the callback is ready to be called, the runtime environment invokes the task with the appropriate arguments. The signature of the callback is determined by the asynchronous function.

For example, in the example above, the callback for the `readFile` procedure is a function with 2 parameters (`err`, `data`) which represent either an error object or the data that was read from the file. These arguments are passed to the callback when the event is signaled.

The callback is a closure which is created in the context of the asynchronous function. Hence, according to the operational semantics of the language, we know it has access to the environment in which it was created (not to the environment which is current when the closure is invoked). See for example how the environment of the callback is used:

Callbacks and the Call Stack

```
const readJSONTime = (filename, callback) => {  
  const invoked = new Date(); // Timestamp when the read is invoked.  
  fs.readFile(filename, "utf8", (err, res) => {  
    if (err) {  
      callback(err, undefined);  
    } else {  
      // This accesses a variable from the closure env.  
      console.log("Invoked at: ", invoked);  
      console.log("Callback at: ", new Date());  
      callback(undefined, JSON.parse(res));  
    }  
  });  
};  
  
readJSONTime("test.async", (err, res) => console.log(res));  
/*  
  Invoked at: 2020-05-23T18:30:37.540Z  
  Callback at: 2020-05-23T18:30:37.548Z  
  { id: 1, test: 'async' }  
*/
```

Even if the **environment** is available, the callback is evaluated in a different **control context** - that is, in a different call stack.

Composing Asynchronous Functions

Composing synchronous functions is easy: given a function $f(x:T_x):T_f$ and a function $g(y:T_y):T_x$ - we can compose the calls by simply passing the return value of g as a parameter to f : $f(g(y))$. As long as the types are compatible, this composition works (the return type of g must be the parameter type of f).

Composing asynchronous functions is more challenging: we must write a specific callback each time we compose two functions:

```
g(y, (gRes) => f(gRes, callback))
```

Composing Asynchronous Functions

If we want to compose three functions - the callbacks must be nested accordingly - instead of `f(g(h(x)))` we must write:

```
h(x, (hRes) => {  
    g(hRes, (gRes) => {  
        f(gRes, callback);  
    });  
});
```

Note how the order of the occurrences has changed: we read the program in the order in which the functions are invoked (first invoked **h**, then invoke **g**, then invoke **f**).

The Type of Asynchronous Functions

When we think of composing synchronous functions, we check their type - that is, we verify that the type returned by **g** matches the type expected by **f** in order to enable the composition **f(g(x))**.

The Type of Asynchronous Functions

Asynchronous functions do not return any value - that is, they are of return type **void**.

Instead, they post a future task which will receive a parameter of a certain type in the future.

The Type of Asynchronous Functions

That is, if a synchronous function has type

`f(x: Tx): Tf` - the corresponding
asynchronous function will have type

```
asyncF(x: Tx,  
       callback: (Tf -> Tc)): void
```

Error Handling with Asynchronous Procedures (Success / Fail Callbacks)

In many cases, asynchronous functions can fail

- this is the case for most I/O calls (file system, networking calls) which are most likely to be used in an asynchronous manner.

Error Handling with Asynchronous Procedures (Success / Fail Callbacks)

To process the case of errors - the callback passed to the async function must be prepared to deal with either a success outcome, or with an error outcome. Conceptually - this means the async function has 2 callbacks: one to process success, one to process failure.

Error Handling with Asynchronous Procedures (Success / Fail Callbacks)

In the examples we reviewed above, we worked with a callback that has two parameters - one for error and one for the returned data in case of success.

Error Handling with Asynchronous Procedures (Success / Fail Callbacks)

When composing asynchronous functions with two parameters of this type, we introduce systematic complexity - because there is no way to “stop early” the chain of calls when an error is detected - as we would do by throwing an exception in a synchronous case. We must handle the error case at all steps of the chain of calls.

Error Handling with Asynchronous Procedures (Success / Fail Callbacks)

For example, assuming the function f , g and h can return an error or a success, the composition of $f(g(h(x)))$ requires the following structure:

Error Handling with Asynchronous Procedures (Success / Fail Callbacks)

```
h(x, (hErr, hRes) => {  
  if (hErr) {  
    failCallback(hErr);  
  } else {  
    g(hRes, (gErr, gRes) => {  
      if (gErr) {  
        failCallback(gErr);  
      } else {  
        f(gRes, callback);  
      }  
    });  
  }  
});
```

Error Handling with Asynchronous Procedures (Success / Fail Callbacks)

The following issues make the usage of asynchronous functions difficult for the programmer:

- all calls must deal with error cases
- sequence of calls or composed calls must be translated into nested callbacks
- the types of the functions are obscure

Promises are a general programming pattern designed to simplify asynchronous composition, in particular error handling.

Using Promises to Simplify Asynchronous Composition

A promise represents the result of an asynchronous operation - it is an object which serves as the proxy for a delayed computation which can be in different states:

- **pending** – The initial state of a promise - before the computation has completed.
- **fulfilled** – The state of a promise representing a successful operation.
- **rejected** – The state of a promise representing a failed operation.

Once a promise is fulfilled or rejected, it becomes immutable (*i.e.* it can never change again).

Promises allow the client (the programmer) to associate handlers with an asynchronous action's eventual success value or failure reason.

Using Promises to Simplify Asynchronous Composition

This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

Using Promises to Simplify Asynchronous Composition

Clients of a promise register with the promise by submitting the callbacks they want the promise to execute. Since we want to simplify error handling, the client of a promise can register one callback for success completion (fulfillment) and one for error handling (rejection handler).

The registration is performed using two methods of the Promise interface:

- `Promise.then(successHandler)`
- `Promise.catch(rejectionHandler)`

The handlers are functions which are submitted to be executed in the future, when the promise is resolved (either fulfilled or rejected).

Using Promises to Simplify Asynchronous Composition

At this point, our understanding of a promise is that it is an object with the following state:

- task: the asynchronous task to be computed
- value: the result of the task when it is resolved
- state: state of the promise (pending, fulfilled or rejected)
- handlers: the handlers for success and failure

The key events in the lifecycle of a promise correspond to state transitions:

Using Promises to Simplify Asynchronous Composition

- created in state Pending
- pending to fulfilled: triggered when the async computation completes (usually by the event loop); triggers the success handlers.

Using Promises to Simplify Asynchronous Composition

- pending to rejected: triggered when the async computation ends in error; triggers the failure handlers.
- attach handlers: if the state is pending, just add the handlers to the internal state of the promise, else immediately invoke the new handler with the stored value of the promise.

Making a Promise

To construct a promise from an asynchronous function with a callback, we use the Promise constructor and abstract the callbacks for success and failure:

```
const readFilePromise = (filename: string): Promise<string> =>
  new Promise<string>((resolve, reject) => {
    fs.readFile(filename, (err, res) => {
      if (err) {
        reject(err);
      } else {
        resolve(res.toString("utf8"));
      }
    });
  });
```

The Type of a Promise

A promise is a container for a value which may become available in the future. A function such as `readFilePromise` above returns a promise - in contrast to the original `readFile` asynchronous function which had a `void` return type.

The Type of a Promise

This ends up simplifying the type and making it more similar to the familiar synchronous version:

```
readFileSync(filename: string): string;
```

```
readFile(filename: string,  
         callback: (err, data:string) -> T): void;
```

```
readFilePromise(filename: string): Promise<string>;
```

TypeScript supports generic Promise types to document this type of return value.

We use the Promisified version of the asynchronous function according to the Promise client pattern:

- We decouple the creation of the promise from its consumption.
- We separate success and error handling in two separate concerns.

To this end, we use the 2 methods of the `Promise` object: `then(successHandler)` and `catch(errorHandler)`:

Using a Promise

```
const testContent = readFilePromise("test.async");
testContent
  .then((content: string) =>
    console.log("Content:", JSON.parse(content)))
  .catch((err) => console.error(err));
```

Note the style that the `then` and `catch` methods implement: they return a value of the same type as the object on which they are invoked, so that they can be chained. This style is called the **fluent interface pattern**.

Using promises, we can achieve 3 main benefits over the structure that callbacks only would require:

Chaining Promises

- The type of functions returning Promises is more informative and similar to the simple types of synchronous versions
- We can chain sequences of asynchronous calls in a chain of `.then()` calls.
- We can aggregate error handling in a single handler for a chain of calls, in a way similar to exception handling.

The following example illustrates these 3 benefits:

- We want to read a file containing a JSON value
- Update the content of the JSON
- Write back the updated value of the JSON to the file

This requires a chain of asynchronous calls
(reading the file and writing it).

Note that the error handler needs to be
specified only once for the two operations

Chaining Promises

```
const readFilePromise = (filename: string): Promise<string> => {
  new Promise<string>((resolve, reject) => {
    fs.readFile(filename, (err, res) => {
      if (err) {
        reject(err);
      } else {
        resolve(res.toString("utf8"));
      }
    });
  });
};
```

Chaining Promises

```
const writeFilePromise =  
  (filename: string, content: string): Promise<void> =>  
    new Promise((resolve, reject) => {  
      fs.writeFile(filename, content, (err) => {  
        if (err) {  
          reject(err);  
        } else {  
          resolve();  
        }  
      });  
    });
```

Chaining Promises

// Chain the calls together

```
const readUpdateWrite = (filename: string): Promise<void> =>
  readFilePromise(filename)
    .then((content) => {
      let j = JSON.parse(content);
      j.lastModified = new Date();
      return writeFilePromise(filename, JSON.stringify(j));
    })
    .catch((err) => console.error(err));
```

Chaining Promises

```
writeFilePromise("promiseExample.async",  
                 JSON.stringify({ a: 1 })))  
  .then(() => console.log("File is created"))  
  .then(() => readFilePromise("promiseExample.async"))  
  .then((content) => console.log(JSON.parse(content)))  
  .then(() => readUpdateWrite("promiseExample.async"))  
  .then(() => console.log("File is updated"))  
  .then(() => readFilePromise("promiseExample.async"))  
  .then((content) => console.log(JSON.parse(content)));
```

Promises simplify the usage of asynchronous functions on three aspects:

- Types of functions that return promises are clearer: for a synchronous function `f(x: T1): T2` the corresponding Promise version will have type

`fp(x: T1): Promise<T2>.`

This is in contrast with a callback-based version which would have type

`fc(x: T1, (err: Error, res:T2) => T3): void.`

- Composition is simplified by chaining `.then(handler)`.
- Error handling can be specified in a single place, as errors are cascaded through promises in a chain.

Promises improve significantly over callback-based asynchronous functions, but still cannot be used as simply as synchronous functions within simple control flow operations (sequence, conditionals, composition). Instead, we still need to pass the promise to a function using the `.then(function)` mechanism.

From Promises to `async` / `await`

Promises are made even easier to use through the mechanism of `async` and `await` syntactic sugar which was introduced in standard JavaScript around 2017. These two new syntactic keywords introduce syntactic variants of the pattern which consists of building a promise and then chaining code into the `then` and `catch` methods of the promise.

So that for example:

```
// Chain the calls together
const readUpdateWrite = (filename: string): Promise<void> => {
  return readFilePromise(filename)
    .then((content) => {
      let j = JSON.parse(content);
      j.lastModified = new Date();
      return writeFilePromise(filename, JSON.stringify(j));
    })
    .catch((err) => console.error(err));
};
```

From Promises to `async / await`

is equivalent to the following syntactic variant:

```
// The async/await version
const readUpdateWrite = async (filename: string): Promise<void> => {
  try {
    const content = await readFilePromise(filename);
    let j = JSON.parse(content);
    j.lastModified = new Date();
    return writeFilePromise(filename, JSON.stringify(j));
  } catch (err) {
    return console.error(err);
  }
};
```

The `specification` for `async` and `await` explains how `async` and `await` are *syntactic abstraction* around Promises.

Key points about using `async` and `await`:

- `async` functions always return a Promise value.
- `await` can only be used within the body of an `async` function.
- `await` is followed by a call that produces a Promise (usually an `async` function)

Key points about using `async` and `await`:

- `await` can throw an exception
(corresponding to the fact that the promise that is `awaited` is rejected) - it should therefore be wrapped in `try/catch` construct.

From Promises to `async` / `await`

`await` <something producing a promise>; <continuation>;

is equivalent to:

<something producing a promise>.then(() => <continuation>);

The following example is from [here](#):

```
function logFetch(url) {  
  return fetch(url)  
    .then((response) => response.text())  
    .then((text) => {  
      console.log(text);  
    })  
    .catch((err) => {  
      console.error("fetch failed", err);  
    });  
}
```

From Promises to `async` / `await`

is equivalent to:

```
async function logFetch(url) {  
  try {  
    const response = await fetch(url);  
    console.log(await response.text());  
  } catch (err) {  
    console.log("fetch failed", err);  
  }  
}
```

JavaScript (and other programming languages, including Python and Scheme) offer a mechanism called generators which can be combined with promises to provide excellent support for asynchronous programming.

Generators are functions which can be exited and later re-entered. Their context (variable bindings and current control locations) will be saved across re-entries. A generator is created with the keyword **function***.

Calling a generator function does not execute its body immediately; an **iterator** object for the function is returned instead. An iterator is a map with two fields:
`{value: any, done: boolean}` and that has a `next()` method:

```
interface IteratorResult {  
    value: any;  
    done: boolean;  
}
```

```
interface Iterator {  
    next(): IteratorResult;  
}
```

When the iterator's `next()` method is called, the generator function's body is executed until the first `yield` expression, which specifies the value to be returned from the iterator.

The `next()` method returns an object with a `value` property containing the yielded value and a `done` property which indicates whether the generator has yielded its last value as a boolean. Calling the `next()` method with an argument will resume the generator function execution, replacing the `yield` statement where execution was paused with the argument from `next()`.

A **return** statement in a generator, when executed, will make the generator's **done** field **true**. If a value is returned, it will be passed back as the value. A generator which has returned will not yield any more values. (That is, **return x** is like **yield x** plus the side effect that the generator is now done).

Let us examine a few examples of using generators:

```
function* idMaker() {  
  let index = 0;  
  while (index < 3) {  
    yield index++;  
  }  
}
```

Generators and Co-routines

```
const gen = idMaker();  
  
console.log(gen.next()); // { value: 0, done: false }  
console.log(gen.next()); // { value: 1, done: false }  
console.log(gen.next()); // { value: 2, done: false }  
console.log(gen.next()); // { value: undefined, done: true }
```

We can also pass a parameter back to the generator by passing it through the `next(val)` call:

Generators and Co-routines

```
function* demo() {  
  const res = yield 10;  
  assert(res === 32);  
  return 42;  
}
```

```
const d = demo();  
console.log(d.next());    // { value: 10, done: false }  
console.log(d.next(32)); // { value: 42, done: true }  
console.log(d.next());    // { value: undefined, done: true }
```

The **function*** special form is used to construct a generator. Within the body of a generator, the **yield** special form can be used.

Generators are most often consumed inside loops - and as their names indicate they generate a sequence of values in a lazy manner: instead of eagerly constructing a list of values, the generator knows how to generate the values only when asked to.

The **for** loop of JavaScript is a syntax which is adapted to consume any object which implements the iterator protocol, and hence works well with generators:

```
function* foo() {  
  yield 1;  
  yield 2;  
  return 3;  
}  
  
for (let v of foo()) {  
  console.log(v);  
}
```

Generators can be used to generate computed sequences - which can even be infinite, since they are only generated when requested:

```
function* range(start, end) {  
  for (let i = start; i < end; i++) {  
    yield i;  
  }  
}
```

```
for (let n of range(1, 5)) {  
  console.log(n);  
}
```

A generator can produce a potentially infinite stream of data. Such a generator is useful in case the consumer of the stream can limit the number of items to actually extract.

The following example demonstrates an infinite generator and a higher order function called **take** which takes as input a number and a generator - and returns a new generator which generates less than n items from the generator:

Generators and Co-routines

```
function* naturalNumbers() {  
  for (let n = 0; ; n++) {  
    yield n;  
  }  
}
```

```
function* take(n, generator) {  
  for (let v of generator) {  
    if (n <= 0) return;  
    n--;  
    yield v;  
  }  
}
```

```
for (let n of take(3, naturalNumbers())) {  
  console.log(n);  
}
```

Generators are an efficient mechanism to combine iterations without copying lists of data at each stage.

Consider the following examples:

Generators and Co-routines

```
function* mapGen(generator, f) {  
  for (let v of generator) {  
    yield f(v);  
  }  
}
```

```
function* filterGen(generator, pred) {  
  for (let v of generator) {  
    if (pred(v)) {  
      yield v;  
    }  
  }  
}
```

Such operators can be combined together - with the advantage that the data is not copied between each stage. Instead, each time an element of the combination is requested, the chain of functions is evaluated:

Generators and Co-routines

```
const evenSquares =  
  filterGen(mapGen(naturalNumbers(), x => x * x),  
            x => x % 2 === 0);  
  
for (let n of take(4, evenSquares)) {  
  console.log(n);  
}
```

Generators and Co-routines

```
const verboseSquare = (x) => {  
  console.log("square", x, "->", x * x);  
  return x * x;  
};  
  
const verboseIsEven = (x) => {  
  console.log("even?", x, "->", x % 2 === 0);  
  return x % 2 === 0;  
};  
  
const evenSquaresVerbose = filterGen(  
  mapGen(nats(), verboseSquare),  
  verboseIsEven  
);  
  
for (let n of take(3, evenSquaresVerbose)) {  
  console.log("evenSquaresVerbose", n);  
}
```

The order in which the functions are executed is
item-by-item: map-square / filter-even / take

Generators and Co-routines

```
square 0 -> 0
even? 0 -> true
evenSquaresVerbose 0
square 1 -> 1
even? 1 -> false
square 2 -> 4
even? 4 -> true
evenSquaresVerbose 4
square 3 -> 9
even? 9 -> false
square 4 -> 16
even? 16 -> true
evenSquaresVerbose 16
square 5 -> 25
even? 25 -> false
square 6 -> 36
even? 36 -> true
```

In contrast, the traditional `map` and `filter` functions allocate a list (an array) to store the intermediary results. The functions are evaluated on all the array at once - square on all the items of the input array first, then even on all the items of the output array.