Control Flow Goodness

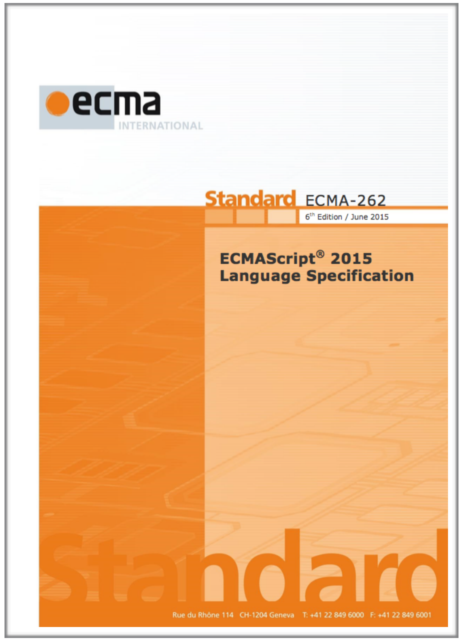
in Modern JavaScript

Tom Van Cutsem

IFIP WGLD 2018

@tvcutsem

Modern JavaScript?



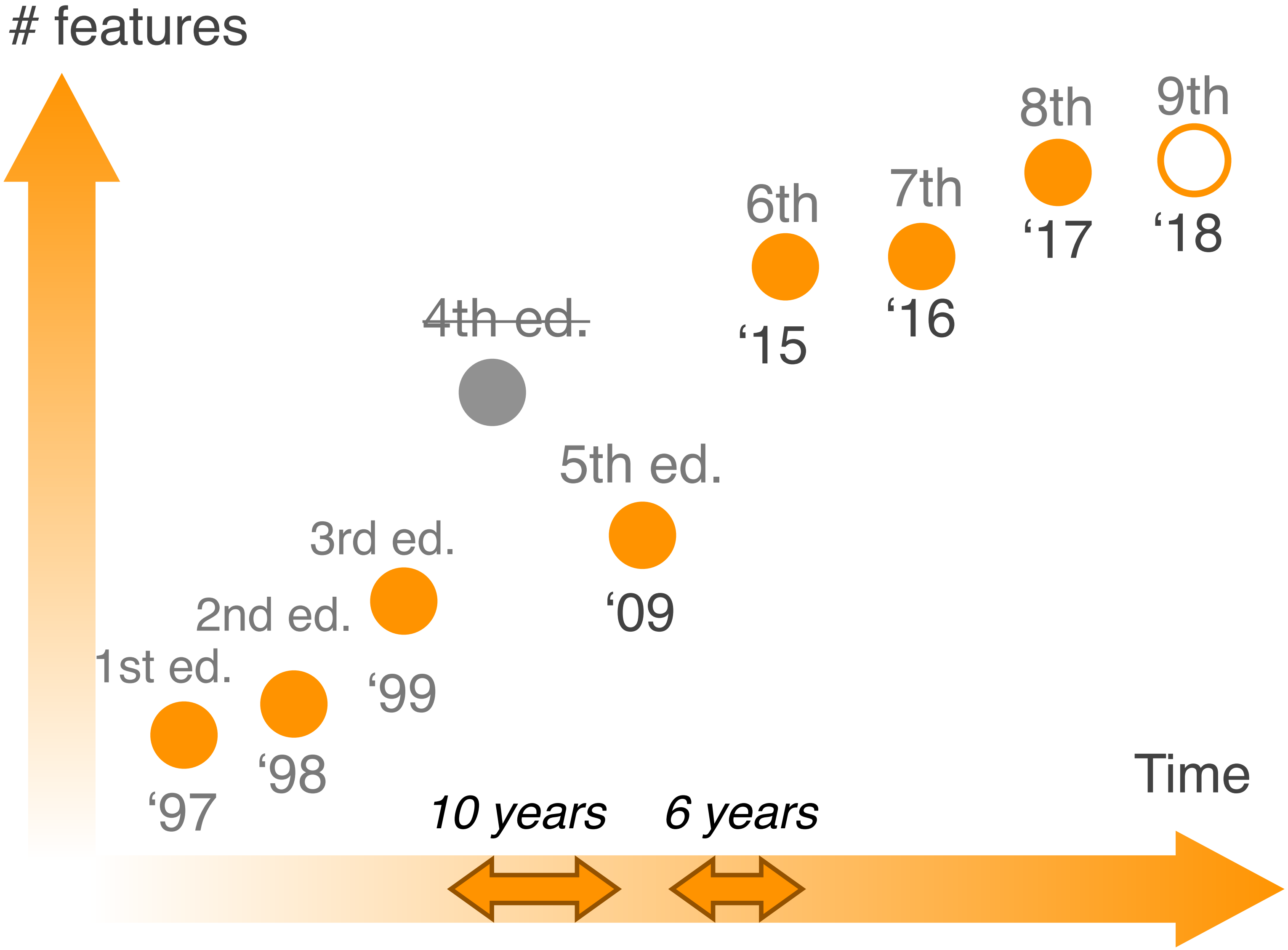
ecma
INTERNATIONAL

Standard ECMA-262
6th Edition / June 2015

ECMAScript® 2015
Language Specification

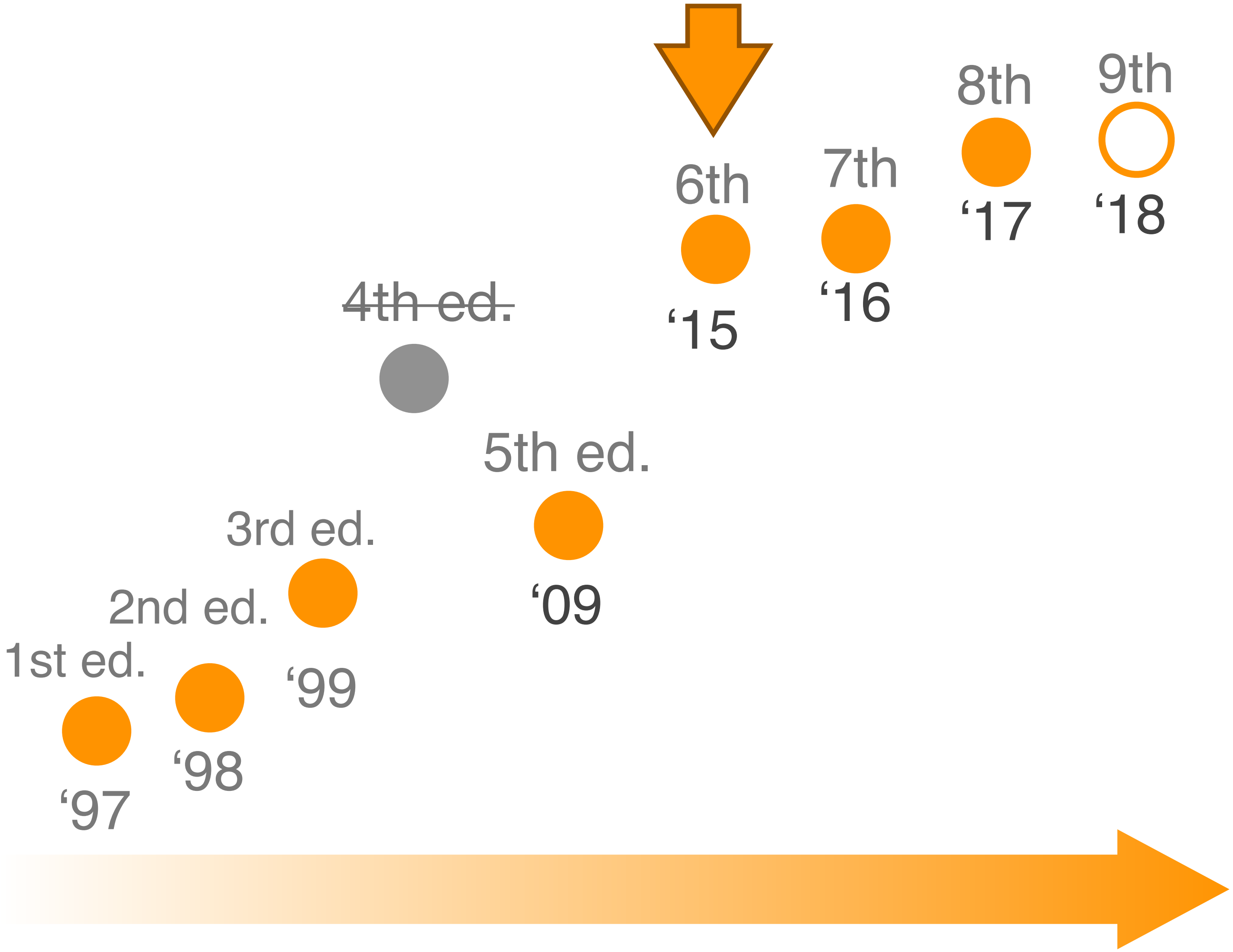
Standard

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001



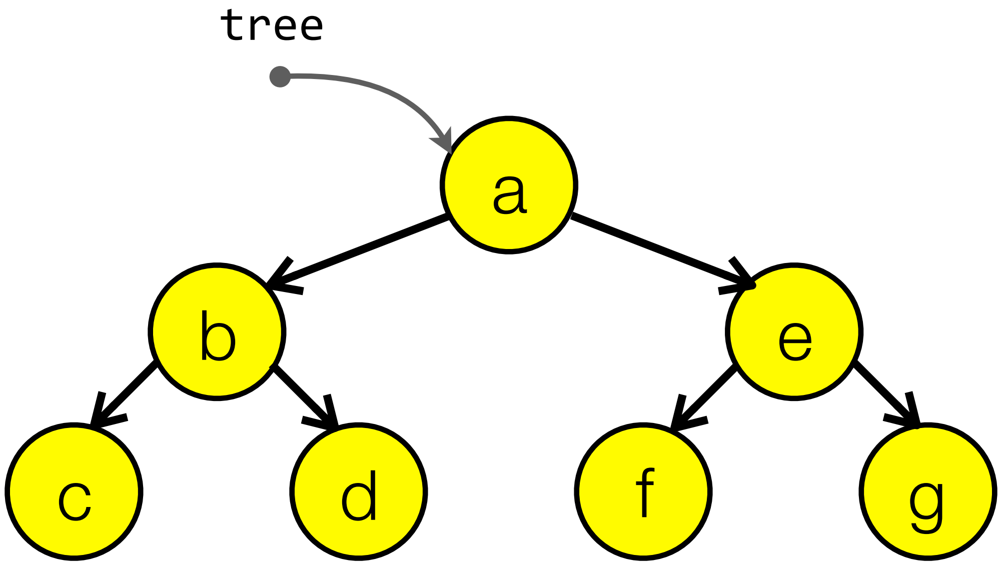
New control flow features in ECMAScript 2015

- Iterators
- Generators
- Promises



Computer Science 101: binary trees

```
interface Tree<T> {
  key    : T,
  left?  : Tree<T>,
  right? : Tree<T>
}


let tree: Tree<string> = {
  key: "a",
  left: {
    key: "b",
    left:  { key: "c" },
    right: { key: "d" }
  },
  right: {
    key: "e",
    left:  { key: "f" },
    right: { key: "g" }
  }
};
```

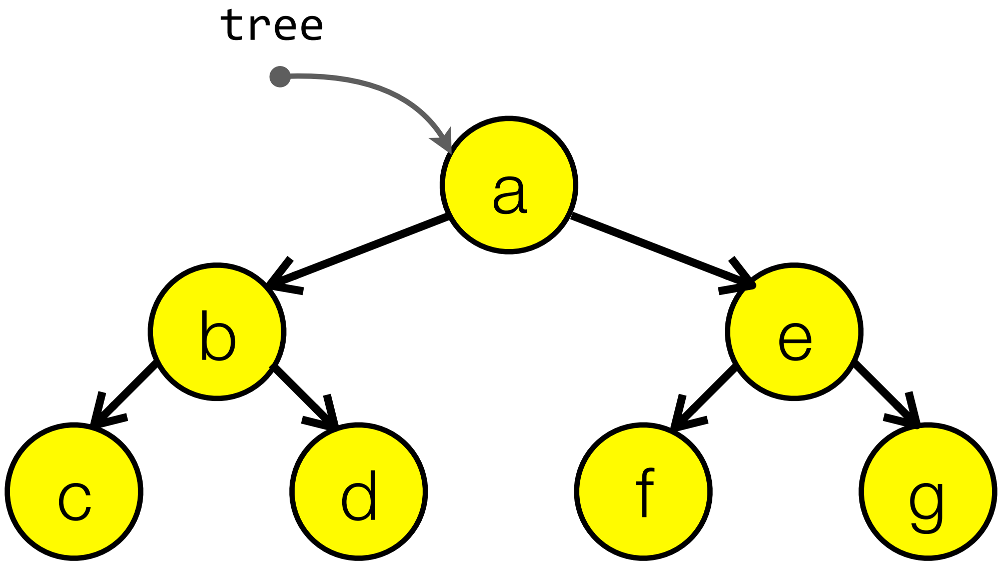


Computer Science 101: pre-order tree walk

- Visit node, then left subtree, then right subtree

```
assert.deepEqual(preOrder(tree), ["a", "b", "c", "d", "e", "f", "g"])
```

```
let tree: Tree<string> = {
  key: "a",
  left: {
    key: "b",
    left:  { key: "c" },
    right: { key: "d" }
  },
  right: {
    key: "e",
    left:  { key: "f" },
    right: { key: "g" }
  }
};
```

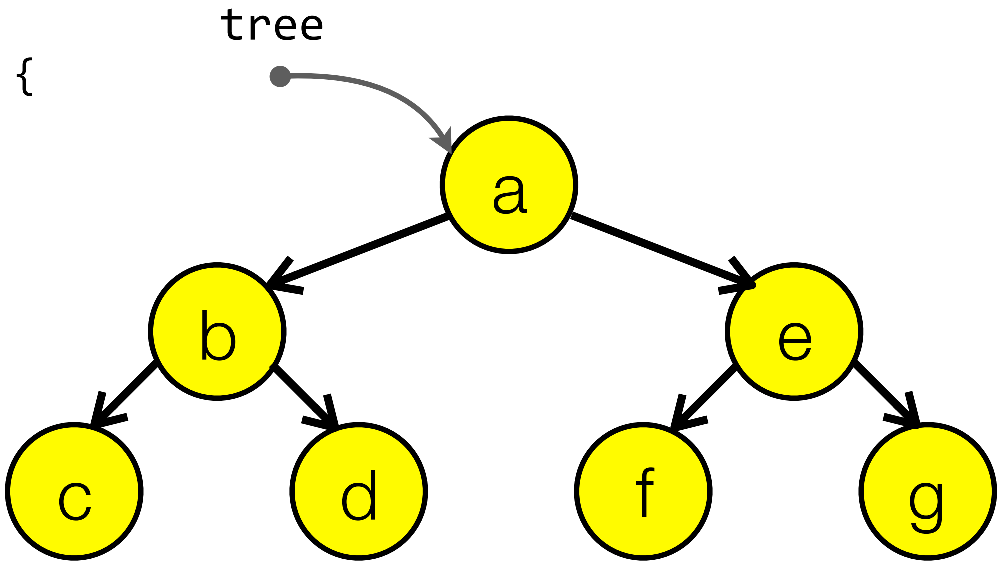


Computer Science 101: pre-order tree walk

- Visit node, then left subtree, then right subtree

```
assert.deepEqual(preOrder(tree), ["a", "b", "c", "d", "e", "f", "g"])
```

```
function preOrder(tree, accum = []) {
  if (tree) {
    accum.push(tree.key);
    preOrder(tree.left, accum);
    preOrder(tree.right, accum);
  }
  return accum;
}
```

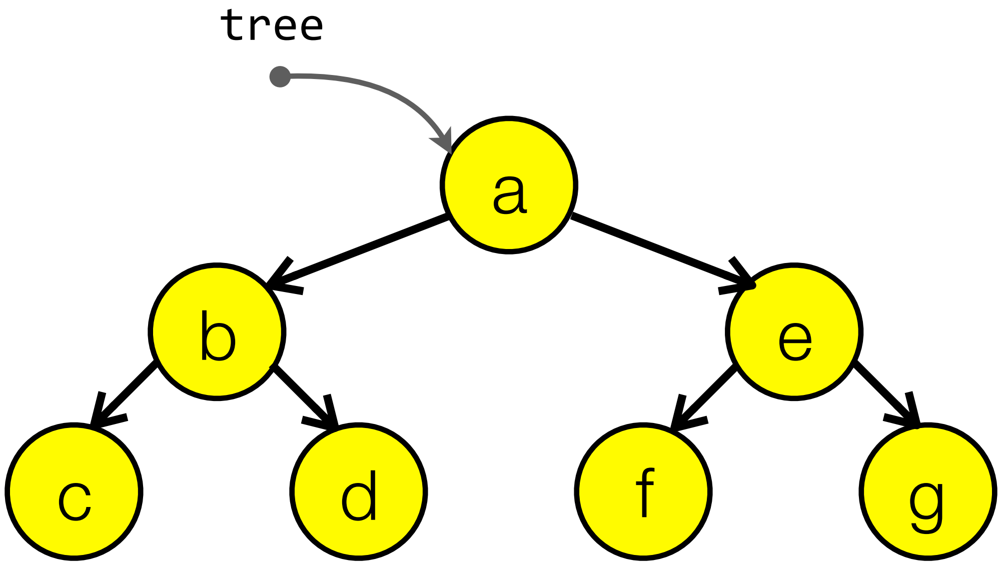


Iterators

- How to support incremental iteration? Change the algorithm so that it returns an **iterator**.

```
function preOrderIter(tree: Tree<T>): Iterator<T>;
```

```
interface Iterator<T> {
  next() : IteratorResult<T>;
}

interface IteratorResult<T> {
  value : T;
  done  : bool;
}
```

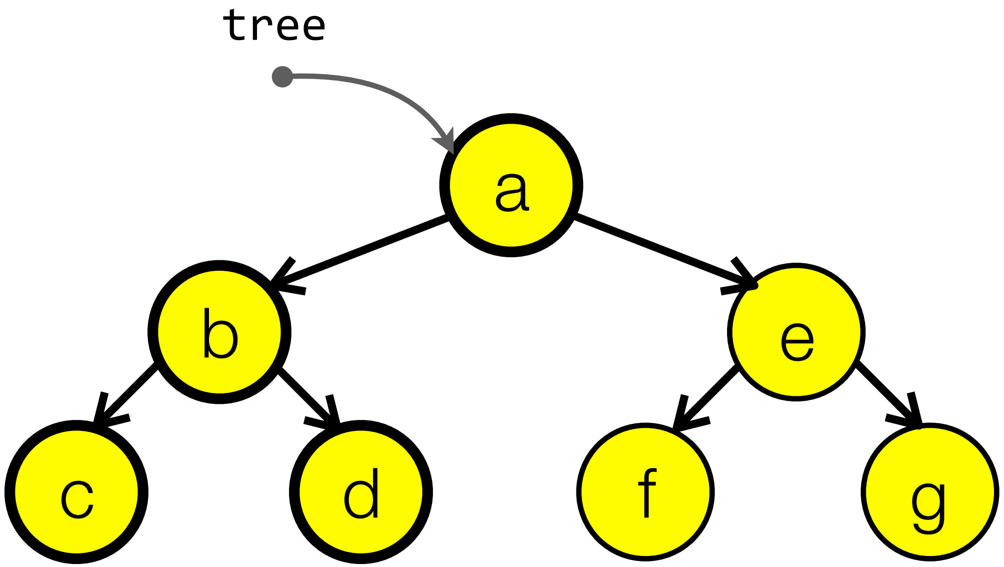


Using Iterators in ECMAScript 5

- Iteration protocol is explicit in the code

```
function preOrderIter(tree: Tree<T>): Iterator<T>;
```

```
let iter = preOrderIter(tree);
let nxt = iter.next();
while (!nxt.done) {
  let k = nxt.value;
  if (k == "d")
    break;
  console.log(k);
  nxt = iter.next();
}
```

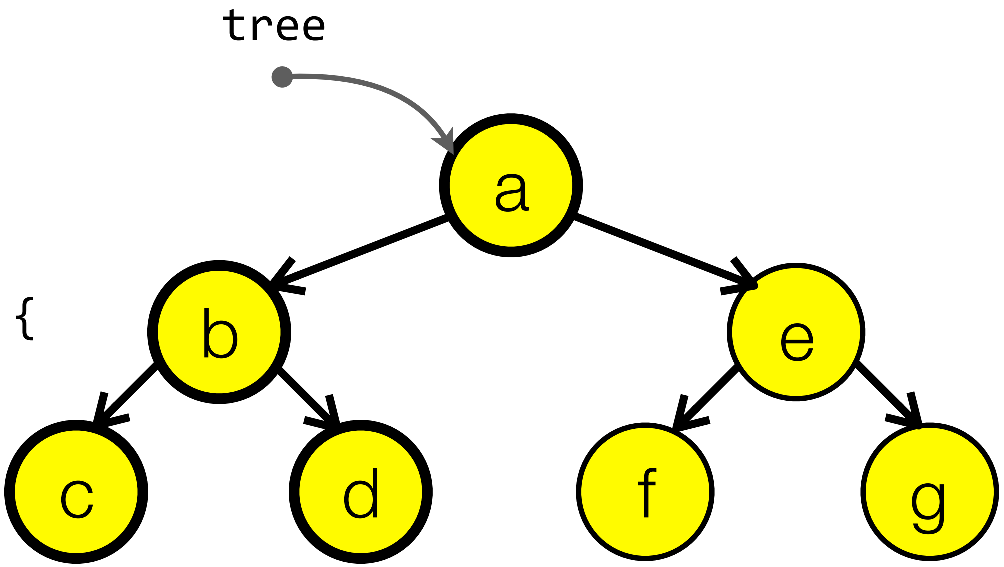


Using Iterators in ECMAScript 6

- New for-of loop enumerates all the elements of an iterator or iterable collection

```
function preOrderIter(tree: Tree<T>): Iterator<T>;
```

```
for (let k of preOrderIter(tree)) {
  if (k == "d")
    break;
  console.log(k);
}
```



Using Iterators in ECMAScript 6

- The iteration protocol is entirely implicit

```
function preOrderIter(tree: Tree<T>): Iterator<T>;
```

ES5

```
let iter = preOrderIter(tree);
let nxt = iter.next();
while (!nxt.done) {
  let k = nxt.value;
  if (k == "d")
    break;
  console.log(k);
  nxt = iter.next();
}
```

ES6

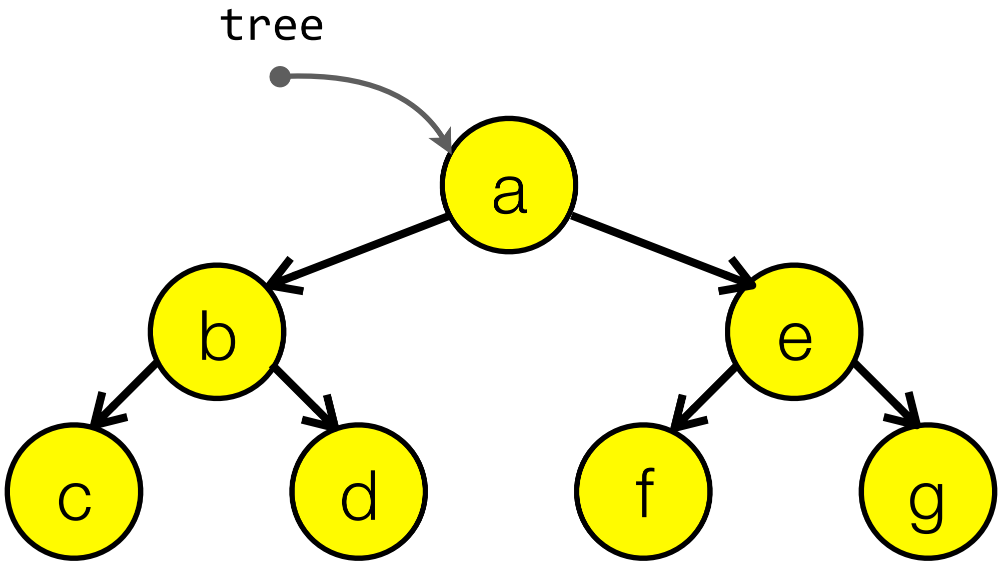
```
for (let k of preOrderIter(tree)) {
  if (k == "d")
    break;
  console.log(k);
}
```

Defining Iterators in ECMAScript 5

- We still need to implement our incremental pre-order tree walk algorithm

```
function preOrderIter(tree: Tree<T>): Iterator<T>;
```

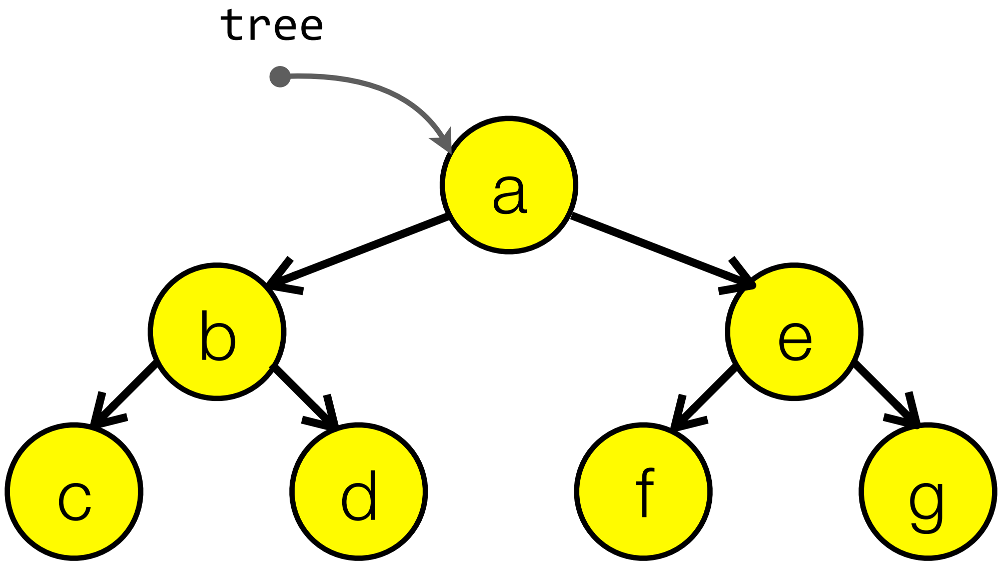
```
interface Iterator<T> {
  next() : IteratorResult<T>;
}

interface IteratorResult<T> {
  value : T;
  done  : bool;
}
```



Defining Iterators in ECMAScript 5

- Iteration protocol is explicit. Execution state (call stack) is explicit. Can't use recursion anymore.

```
function preOrderIter(tree) {
  let todo = [];
  if (tree) {
    todo.push(tree);
  }
  return {
    next() {
      if (todo.length === 0) {
        return {done: true};
      } else {
        let top = todo.pop();
        if (top.right) {
          todo.push(top.right);
        }
        if (top.left) {
          todo.push(top.left);
        }
        return {done: false, value: top.key};
      }
    }
  };
}
```



Defining Iterators in ECMAScript 5

- Can we have our cake and eat it too?

Elegant but batch

```
function preOrder(tree, accum = []) {
  if (tree) {
    accum.push(tree.key);
    preOrder(tree.left, accum);
    preOrder(tree.right, accum);
  }
  return accum;
}
```

Hairy but incremental

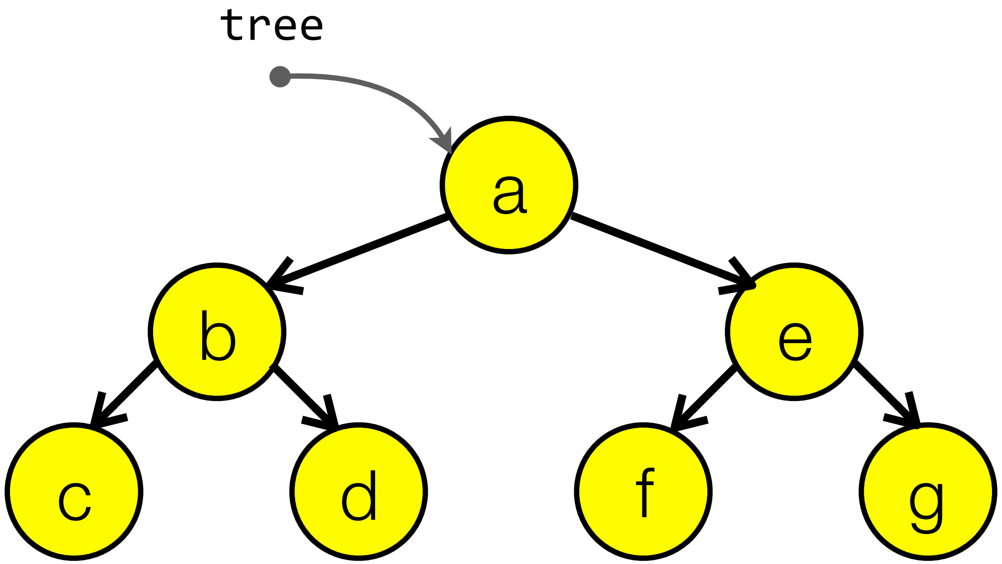
```
function preOrderIter(tree) {
  let todo = [];
  if (tree) {
    todo.push(tree);
  }
  return {
    next() {
      if (todo.length === 0) {
        return {done: true};
      } else {
        let top = todo.pop();
        if (top.right) {
          todo.push(top.right);
        }
        if (top.left) {
          todo.push(top.left);
        }
        return {done: false, value: top.key};
      }
    }
  };
}
```

Generators to the rescue!

- A generator function implicitly creates and returns an iterator

```
function preOrderIter(tree: Tree<T>): Iterator<T>;
```

```
function* preOrderIter(tree) {
  if (tree) {
    yield tree.key;
    yield* preOrderIter(tree.left);
    yield* preOrderIter(tree.right);
  }
}
```



Generators in ECMAScript 6

- Both iteration protocol and execution state become implicit

ES5

Hairy but incremental

```
function preOrderIter(tree) {
  let todo = [];
  if (tree) {
    todo.push(tree);
  }
  return {
    next() {
      if (todo.length === 0) {
        return {done: true};
      } else {
        let top = todo.pop();
        if (top.right) {
          todo.push(top.right);
        }
        if (top.left) {
          todo.push(top.left);
        }
        return {done: false, value: top.key};
      }
    }
  };
}
```

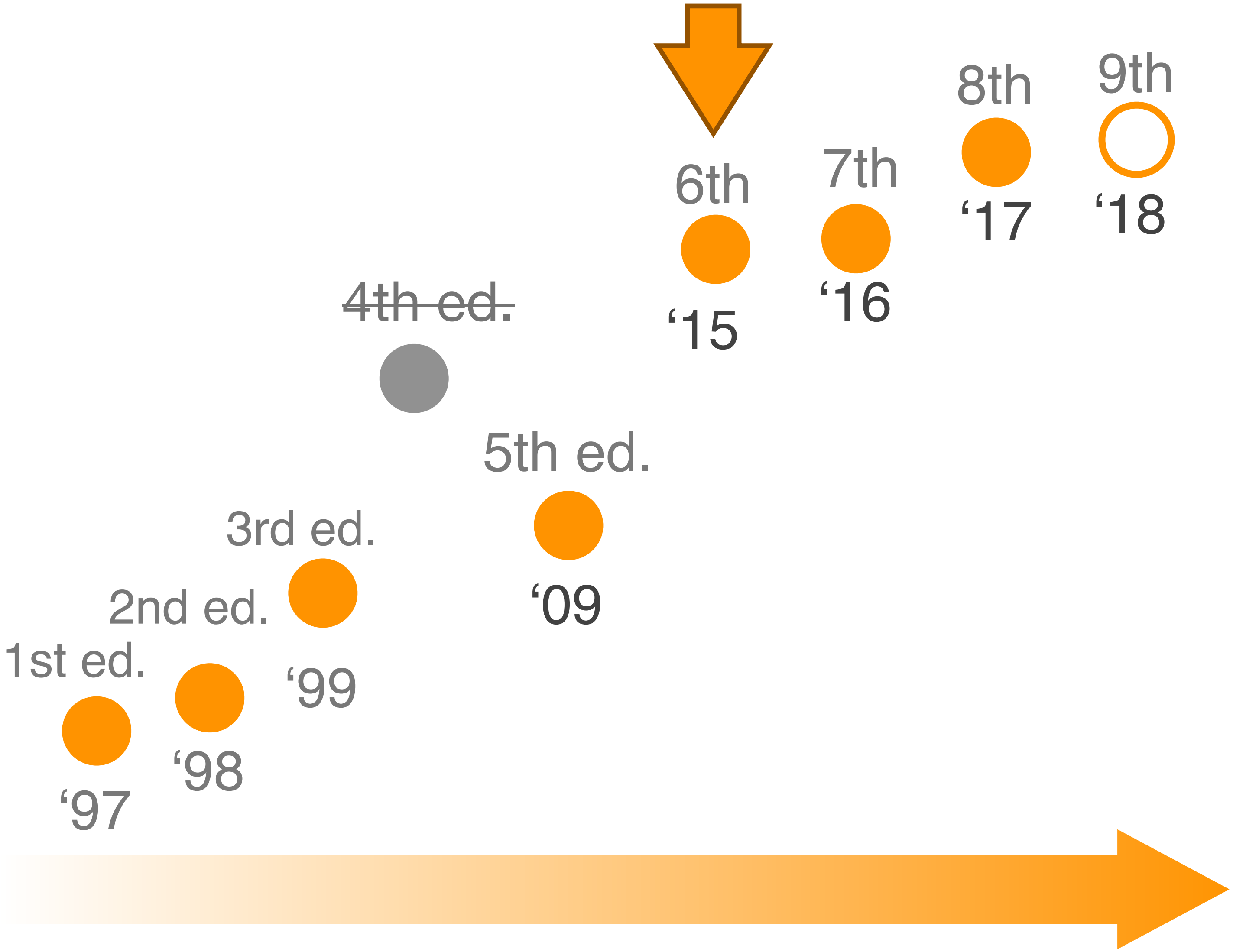
ES6

Elegant *and* incremental

```
function* preOrderIter(tree) {
  if (tree) {
    yield tree.key;
    yield* preOrderIter(tree.left);
    yield* preOrderIter(tree.right);
  }
}
```

New control flow features in ECMAScript 2015

- Iterators
- Generators
- **Promises**



ECMAScript 6 Promises

- A promise is a placeholder for a value that may only be available in the future

ES5

```
readFile(“hello.txt”, function (err, content) {
  if (err) {
    // handle error
  } else {
    // use content
  }
})
```

ES6

```
var pContent = readFile(“hello.txt”);
pContent.then(function (content) {
  // use content
}, function (err) {
  // handle error
});
```

ECMAScript 6 Promises

- A promise is a placeholder for a value that may only be available in the future

ES5

```
readFile(“hello.txt”, function (err, content) {
  if (err) {
    // handle error
  } else {
    // use content
  }
})
```

⬇

ES6

```
var pContent = readFile(“hello.txt”);
var p2 = pContent.then(function (content) {
  // use content
}, function (err) {
  // handle error
});
```

ECMAScript 6 Promises

- Promises can be *chained* to avoid callback hell

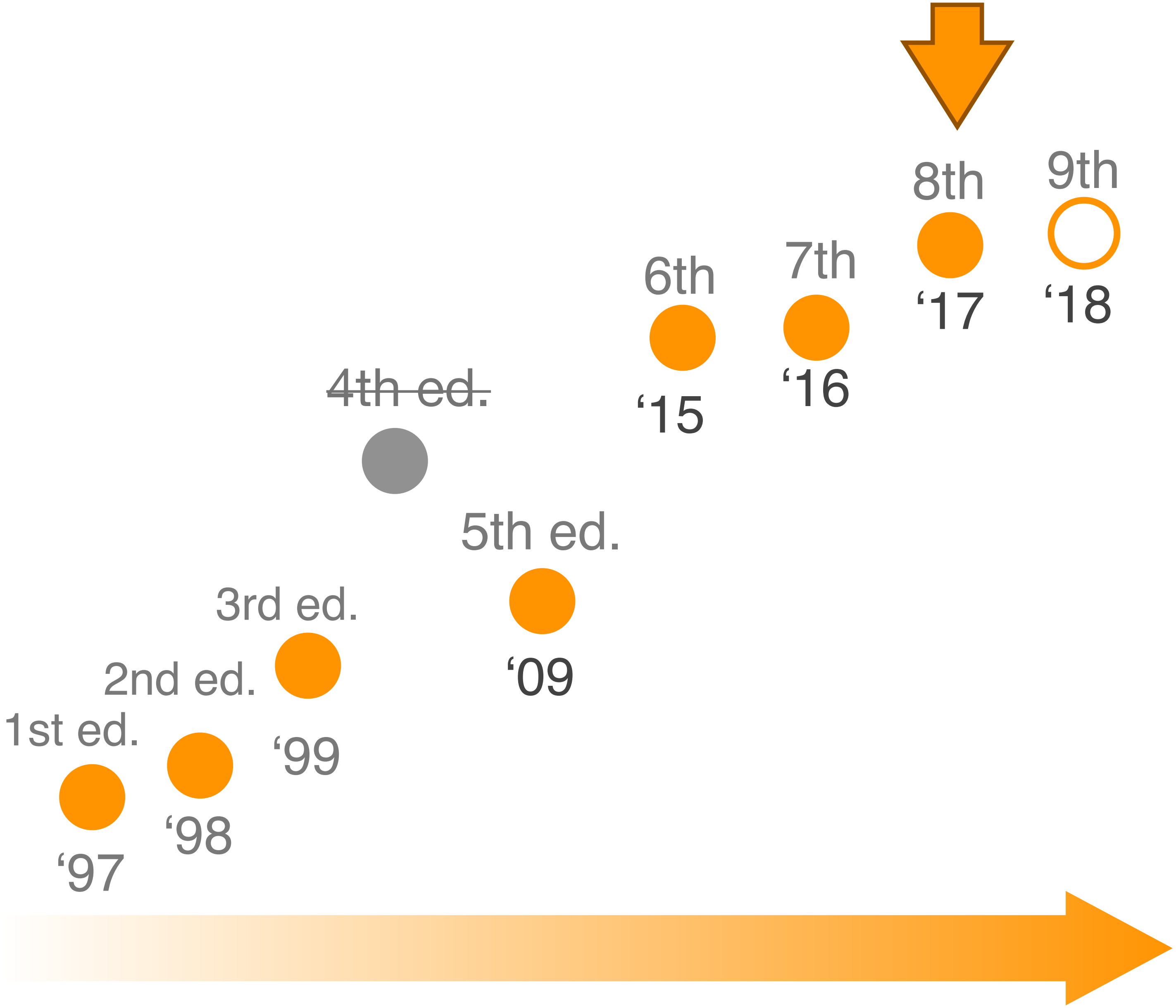
```
function step1(value, callback): void;

step1(function (e,value1) {
    if (e) { return handleError(e); }
    step2(value1, function(e,value2) {
        if (e) { return handleError(e); }
        step3(value2, function(e,value3) {
            if (e) { return handleError(e); }
            step4(value3, function(e,value4) {
                if (e) { return handleError(e); }
                // do something with value4
            });
        });
    });
});
```

```
function step1(value): Promise;

step1(value)
.then(step2)
.then(step3)
.then(step4)
.then(function (value4) {
    // do something with value4
})
.catch(function (error) {
    // handle any error here
});
```

Example adapted from https://github.com/kriskowal/q

New control flow features in ECMAScript 2017

- Async functions



async functions in ECMAScript **2017**

- A C# 5.0 feature that enables asynchronous programming using “direct style” control flow (i.e. no callbacks)

ES6

```
function step1(value): Promise;

step1(value)
.then(step2)
.then(step3)
.then(step4)
.then(function (value4) {
    // do something with value4
})
.catch(function (error) {
    // handle any error here
});
```

ES2017

```
function step1(value): Promise;

(async function() {
  try {
    var value1 = await step1();
    var value2 = await step2(value1);
    var value3 = await step3(value2);
    var value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
}())
```

Duality between async functions and generators

- Generators can be used as async functions, with some tinkering
- There exist libraries that transform async functions into generators

ES2017

```
(async function() {
  try {
    var value1 = await step1();
    var value2 = await step2(value1);
    var value3 = await step3(value2);
    var value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
}())
```

ES2015

```
co(function*() {
  try {
    var value1 = yield step1();
    var value2 = yield step2(value1);
    var value3 = yield step3(value2);
    var value4 = yield step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
})
```

async functions in ECMAScript **5** (!)

- Babel plug-in based on Facebook Regenerator

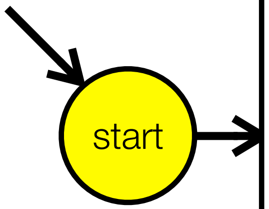
facebook.github.io/regenerator

- Also in TypeScript 1.7+

github.com/lukehoban/ecmascript-asyncawait

ES2017

```
(async function() {
  try {
    var value1 = await step1();
    var value2 = await step2(value1);
    var value3 = await step3(value2);
    var value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
}())
```

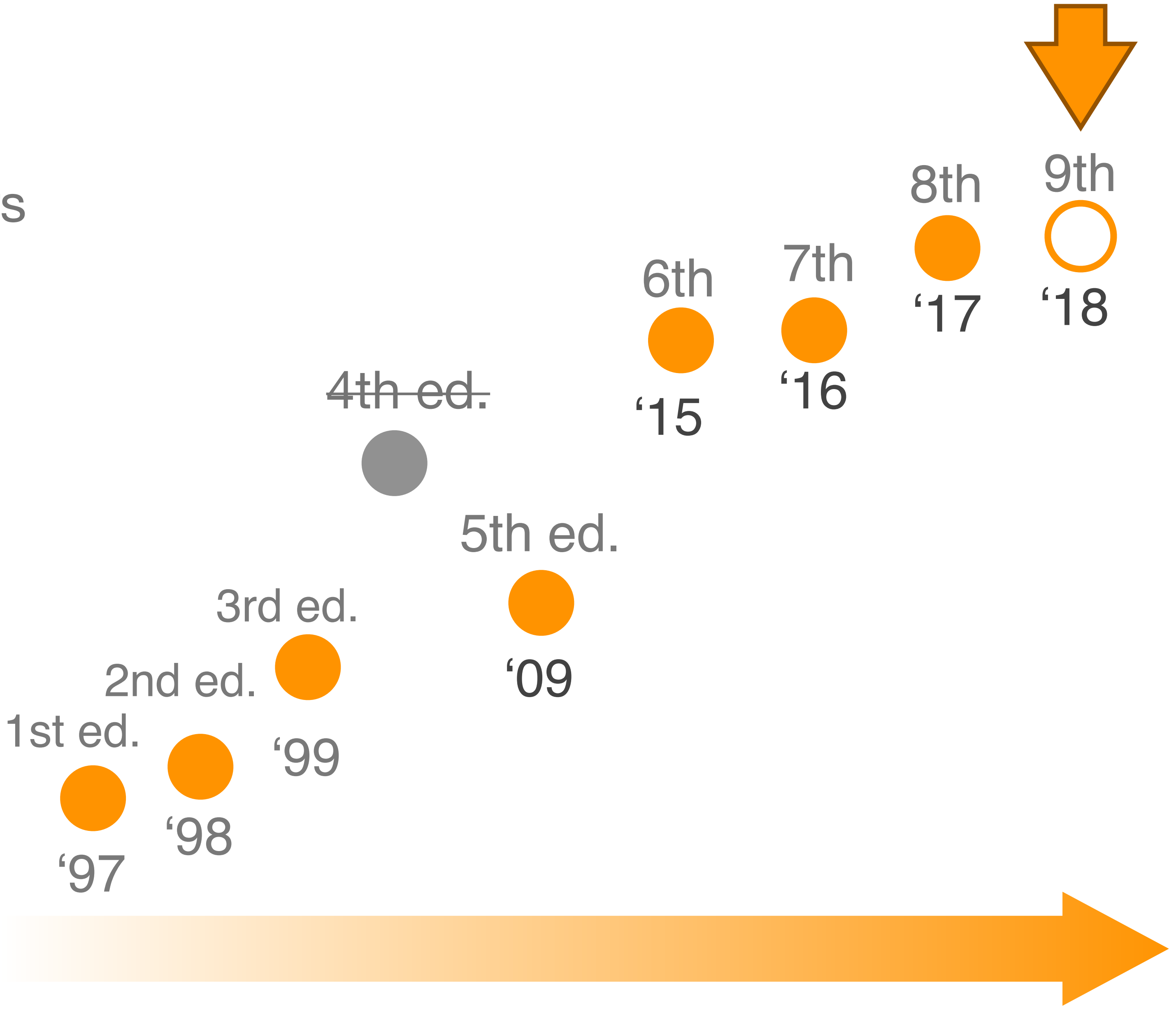


ES5

```
(function callee$0$0() {
  var value1, value2, value3, value4;
  return regeneratorRuntime.async(function ca
    while (1) switch (context$1$0.prev = cont
      case 0:
        context$1$0.prev = 0;
        context$1$0.next = 3;
        return regeneratorRuntime.awrap(step1
      …

})();
```

New control flow features in ECMAScript 2018

- Async iterators
- Async generators



1st ed. ‘97

2nd ed. ‘98

3rd ed. ‘99

~~4th ed.~~

5th ed. ‘09

6th ‘15

7th ‘16

8th ‘17

9th ‘18

Async Iterators

- ES6 iterator and generator protocols are synchronous…
- …but many Iterable sources are asynchronous in JS

```
interface Iterator<T> {
  next() : IteratorResult<T>;
}

interface IteratorResult<T> {
  value : T;
  done  : bool;
}
```

⇨

```
interface AsyncIterator<T> {
  next() : Promise<IteratorResult<T>>;
}
```

https://github.com/tc39/proposal-async-iteration

Async Iterators

- Async for-of loop can be used in an async function to consume an async iterator

```
function readLines(path: string): AsyncIterator<string>;

async function printLines() {
  for await (let line of readLines(filePath)) {
    print(line);
  }
}
```

Async Generators

- Async generators can await, and yield promises

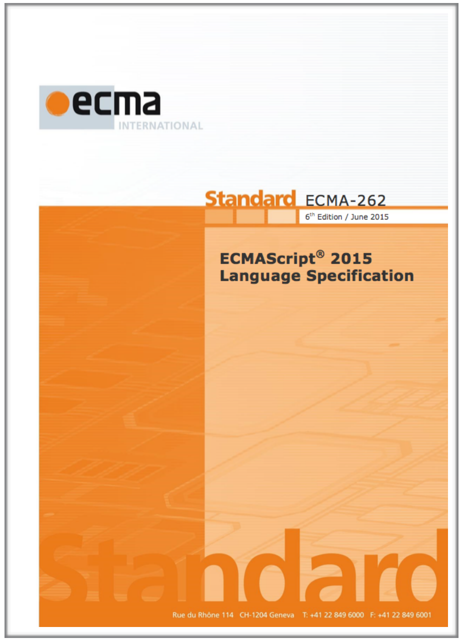
```
function readLines(path: string): AsyncIterator<string>;

async function* readLines(path) {

  let file = await fileOpen(path);

  try {
    while (!file.EOF) {
      yield file.readLine();
    }
  } finally {
    await file.close();
  }
}
```

Async Generators

- What generators are to functions, async generators are to async functions

returns	**Sync**	**Async**
function	`T`	`Promise<T>`
function*	`Iterator<T>`	`AsyncIterator<T>`

Wrap-up: new cflow in Modern JavaScript



ecma INTERNATIONAL

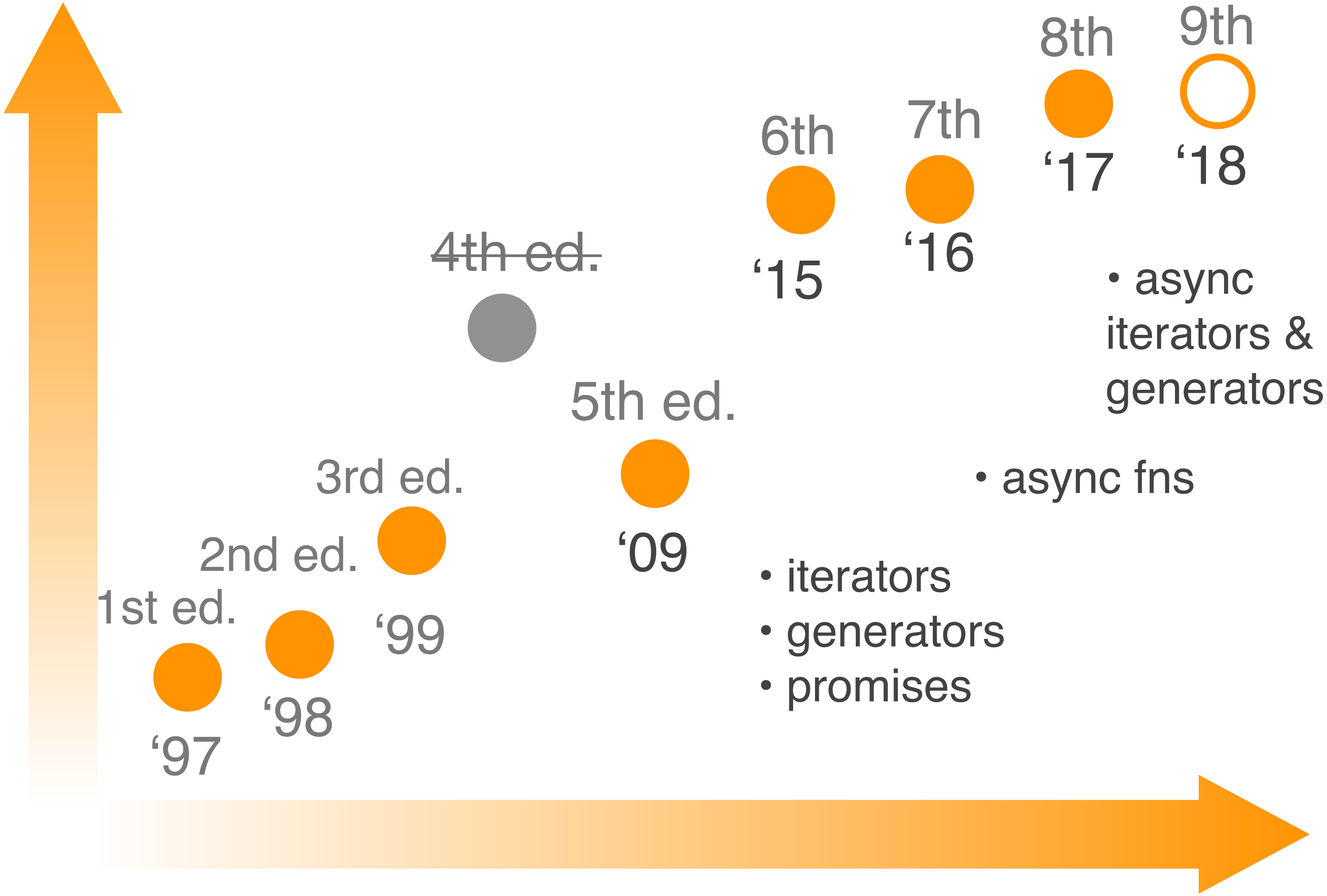
Standard ECMA-262

6th Edition / June 2015

ECMAScript® 2015 Language Specification

Standard

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001



- iterators
- generators
- promises

- async fns

- async iterators & generators

“Callback Hell” has become the “Promised Land”