CJS Module Lexer

build passing

A very fast JS CommonJS module syntax lexer used to detect the most likely list of named exports of a CommonJS module.

Outputs the list of named exports (`exports.name = ...`) and possible module reexports (`module.exports = require('...')`), including the common transpiler variations of these cases.

Forked from https://github.com/guybedford/es-module-lexer.

Comprehensively handles the JS language grammar while remaining small and fast. - ~90ms per MB of JS cold and ~15ms per MB of JS warm, see benchmarks for more info.

Usage

```
npm install cjs-module-lexer
```

For use in CommonJS:

```
const { parse } = require('cjs-module-lexer');

// `init` return a promise for parity with the ESM API, but you do not have to call it

const { exports, reexports } = parse(`
  // named exports detection
  module.exports.a = 'a';
  (function () {
    exports.b = 'b';
  })();
  Object.defineProperty(exports, 'c', { value: 'c' });
  /* exports.d = 'not detected'; */

  // reexports detection
  if (maybe) module.exports = require('./dep1.js');
  if (another) module.exports = require('./dep2.js');

  // literal exports assignments
  module.exports = { a, b: c, d, 'e': f }

  // __esModule detection
  Object.defineProperty(module.exports, '__esModule', { value: true })
`);

// exports === ['a', 'b', 'c', '__esModule']
// reexports === ['./dep1.js', './dep2.js']
```

When using the ESM version, Wasm is supported instead:

```js
import { parse, init } from 'cjs-module-lexer';
// init needs to be called and waited upon
await init();
const { exports, reexports } = parse(source);
```

The Wasm build is around 1.5x faster and without a cold start.

Grammar

CommonJS exports matches are run against the source token stream.

The token grammar is:

```
IDENTIFIER: As defined by ECMA-262, without support for identifier `\` escapes,
filtered to remove strict reserved words:
            "implements", "interface", "let", "package", "private", "protected",
"public", "static", "yield", "enum"

STRING_LITERAL: A `"` or `'` bounded ECMA-262 string literal.

MODULE_EXPORTS: `module` `.` `exports`

EXPORTS_IDENTIFIER: MODULE_EXPORTS_IDENTIFIER | `exports`

EXPORTS_DOT_ASSIGN: EXPORTS_IDENTIFIER `.` IDENTIFIER `=`

EXPORTS_LITERAL_COMPUTED_ASSIGN: EXPORTS_IDENTIFIER `[` STRING_LITERAL `]` `=`

EXPORTS_LITERAL_PROP: (IDENTIFIER  (`:` IDENTIFIER)?) | (STRING_LITERAL `:`
IDENTIFIER)

EXPORTS_SPREAD: `...` (IDENTIFIER | REQUIRE)

EXPORTS_MEMBER: EXPORTS_DOT_ASSIGN | EXPORTS_LITERAL_COMPUTED_ASSIGN

EXPORTS_DEFINE: `Object` `.` `defineProperty `(` EXPORTS_IDENFITIER `,` STRING_LITERAL

EXPORTS_DEFINE_VALUE: EXPORTS_DEFINE `, {`
  (`enumerable: true,`)?
  (
    `value:` |
    `get` (`: function` IDENTIFIER? )?  `() {` return IDENTIFIER (`.` IDENTIFIER | `[`
STRING_LITERAL `]`)? `;`? `}` `,`?
  )
  `})`

EXPORTS_LITERAL: MODULE_EXPORTS `=` `{` (EXPORTS_LITERAL_PROP | EXPORTS_SPREAD) `,`)+
`}`

REQUIRE: `require` `(` STRING_LITERAL `)`

EXPORTS_ASSIGN: (`var` | `const` | `let`) IDENTIFIER `=` (`_interopRequireWildcard
(`)? REQUIRE

MODULE_EXPORTS_ASSIGN: MODULE_EXPORTS `=` REQUIRE

EXPORT_STAR: (`__export` | `__exportStar`) `(` REQUIRE

EXPORT_STAR_LIB: `Object.keys(` IDENTIFIER$1 `).forEach(function (` IDENTIFIER$2 `) {`
  (
    (
      `if (` IDENTIFIER$2 `===` ( `'default'` | `"default"` ) `||` IDENTIFIER$2 `===`
( '__esModule' | `"__esModule"` ) `) return` `;`?
      (
        (`if (Object` `.prototype`? `.hasOwnProperty.call(`  IDENTIFIER `, `
IDENTIFIER$2 `)) return` `;`?)?
        (`if (` IDENTIFIER$2 `in` EXPORTS_IDENTIFIER `&&` EXPORTS_IDENTIFIER `[`
IDENTIFIER$2 `] ===` IDENTIFIER$1 `[` IDENTIFIER$2 `]) return` `;`)?
      )?
    ) |
    `if (` IDENTIFIER$2 `!==` ( `'default'` | `"default"` ) (`&& !` (`Object`
`.prototype`? `.hasOwnProperty.call(`  IDENTIFIER `, ` IDENTIFIER$2 `)` | IDENTIFIER
`.hasOwnProperty(` IDENTIFIER$2 `)`))? `)`
  )
  (
    EXPORTS_IDENTIFIER `[` IDENTIFIER$2 `] =` IDENTIFIER$1 `[` IDENTIFIER$2 `]` `;`? |
    `Object.defineProperty(` EXPORTS_IDENTIFIER `, ` IDENTIFIER$2 `, { enumerable:
true, get` (`: function` IDENTIFIER? )?  `() { return ` IDENTIFIER$1 `[` IDENTIFIER$2
`]` `;`? `}` `,`? `})` `;`?
  )
  `})`
```

Spacing between tokens is taken to be any ECMA-262 whitespace, ECMA-262 block comment or ECMA-262 line comment.

- The returned export names are taken to be the combination of:
  1. All `IDENTIFIER` and `STRING_LITERAL` slots for `EXPORTS_MEMBER` and `EXPORTS_LITERAL` matches.
  2. The first `STRING_LITERAL` slot for all `EXPORTS_DEFINE_VALUE` matches where that same string is not an `EXPORTS_DEFINE` match that is not also an `EXPORTS_DEFINE_VALUE` match.
- The reexport specifiers are taken to be the combination of:
  1. The `REQUIRE` matches of the last matched of either `MODULE_EXPORTS_ASSIGN` or `EXPORTS_LITERAL`.
  2. All *top-level* `EXPORT_STAR` `REQUIRE` matches and `EXPORTS_ASSIGN` matches whose `IDENTIFIER` also matches the first `IDENTIFIER` in `EXPORT_STAR_LIB`.

Parsing Examples

Named Exports Parsing

The basic matching rules for named exports are `exports.name`, `exports['name']` or `Object.defineProperty(exports, 'name', ...)`. This matching is done without scope analysis and regardless of the expression position:

```
// DETECTS EXPORTS: a, b
(function (exports) {
  exports.a = 'a';
  exports['b'] = 'b';
})(exports);
```

Because there is no scope analysis, the above detection may overclassify:

```
// DETECTS EXPORTS: a, b, c
(function (exports, Object) {
  exports.a = 'a';
  exports['b'] = 'b';
  if (false)
    exports.c = 'c';
})(NOT_EXPORTS, NOT_OBJECT);
```

It will in turn underclassify in cases where the identifiers are renamed:

```
// DETECTS: NO EXPORTS
(function (e) {
  e.a = 'a';
  e['b'] = 'b';
})(exports);
```

Getter Exports Parsing

`Object.defineProperty` is detected for specifically value and getter forms returning an identifier or member expression:

```
// DETECTS: a, b, c, d, __esModule
Object.defineProperty(exports, 'a', {
  enumerable: true,
  get: function () {
    return q.p;
  }
});
Object.defineProperty(exports, 'b', {
  enumerable: true,
  get: function () {
    return q['p'];
  }
});
Object.defineProperty(exports, 'c', {
  enumerable: true,
  get () {
    return b;
  }
});
Object.defineProperty(exports, 'd', { value: 'd' });
Object.defineProperty(exports, '__esModule', { value: true });
```

Value properties are also detected specifically:

```
Object.defineProperty(exports, 'a', {
  value: 'no problem'
});
```

To avoid matching getters that have side effects, any getter for an export name that does not support the forms above will opt-out of the getter matching:

```
// DETECTS: NO EXPORTS
Object.defineProperty(exports, 'a', {
  get () {
    return 'nope';
  }
});

if (false) {
  Object.defineProperty(module.exports, 'a', {
    get () {
      return dynamic();
    }
  })
}
```

Alternative object definition structures or getter function bodies are not detected:

```
// DETECTS: NO EXPORTS
Object.defineProperty(exports, 'a', {
  enumerable: false,
  get () {
    return p;
  }
});
Object.defineProperty(exports, 'b', {
  configurable: true,
  get () {
    return p;
  }
});
Object.defineProperty(exports, 'c', {
  get: () => p
});
Object.defineProperty(exports, 'd', {
  enumerable: true,
  get: function () {
    return dynamic();
  }
});
Object.defineProperty(exports, 'e', {
  enumerable: true,
  get () {
    return 'str';
  }
});
```

`Object.defineProperties` is also not supported.

Exports Object Assignment

A best-effort is made to detect `module.exports` object assignments, but because this is not a full parser, arbitrary expressions are not handled in the object parsing process.

Simple object definitions are supported:

```
// DETECTS EXPORTS: a, b, c
module.exports = {
  a,
  'b': b,
  c: c,
  ...d
};
```

Object properties that are not identifiers or string expressions will bail out of the object detection, while spreads are ignored:

```
// DETECTS EXPORTS: a, b
module.exports = {
  a,
  ...d,
  b: require('c'),
  c: "not detected since require('c') above bails the object detection"
}
```

`Object.defineProperties` is not currently supported either.

module.exports reexport assignment

Any `module.exports = require('mod')` assignment is detected as a reexport, but only the last one is returned:

```
// DETECTS REEXPORTS: c
module.exports = require('a');
(module => module.exports = require('b'))(NOT_MODULE);
if (false) module.exports = require('c');
```

This is to avoid over-classification in Webpack bundles with externals which include `module.exports = require('external')` in their source for every external dependency.

In exports object assignment, any spread of `require()` are detected as multiple separate reexports:

```
// DETECTS REEXPORTS: a, b
module.exports = require('ignored');
module.exports = {
  ...require('a'),
  ...require('b')
};
```

Transpiler Re-exports

For named exports, transpiler output works well with the rules described above.

But for star re-exports, special care is taken to support common patterns of transpiler outputs from Babel and TypeScript as well as bundlers like RollupJS. These reexport and star reexport patterns are restricted to only be detected at the top-level as provided by the direct output of these tools.

For example, `export * from 'external'` is output by Babel as:

```
"use strict";

exports.__esModule = true;

var _external = require("external");

Object.keys(_external).forEach(function (key) {
  if (key === "default" || key === "__esModule") return;
  exports[key] = _external[key];
});
```

Where the `var _external = require("external")` is specifically detected as well as the `Object.keys(_external)` statement, down to the exact for of that entire expression including minor variations of the output. The `_external` and `key` identifiers are carefully matched in this detection.

Similarly for TypeScript, `export * from 'external'` is output as:

```
"use strict";
function __export(m) {
    for (var p in m) if (!exports.hasOwnProperty(p)) exports[p] = m[p];
}
Object.defineProperty(exports, "__esModule", { value: true });
__export(require("external"));
```

Where the `__export(require("external"))` statement is explicitly detected as a reexport, including variations `tslib.__export` and `__exportStar`.

Environment Support

Node.js 10+, and [all browsers with Web Assembly support](#).

JS Grammar Support

- Token state parses all line comments, block comments, strings, template strings, blocks, parens and punctuators.
- Division operator / regex token ambiguity is handled via backtracking checks against punctuator prefixes, including closing brace or paren backtracking.
- Always correctly parses valid JS source, but may parse invalid JS source without errors.

Benchmarks

Benchmarks can be run with `npm run bench`.

Current results:

JS Build:

```
Module load time
> 4ms
Cold Run, All Samples
test/samples/*.js (3635 KiB)
> 299ms

Warm Runs (average of 25 runs)
test/samples/angular.js (1410 KiB)
> 13.96ms
test/samples/angular.min.js (303 KiB)
> 4.72ms
test/samples/d3.js (553 KiB)
> 6.76ms
test/samples/d3.min.js (250 KiB)
> 4ms
test/samples/magic-string.js (34 KiB)
> 0.64ms
test/samples/magic-string.min.js (20 KiB)
> 0ms
test/samples/rollup.js (698 KiB)
> 8.48ms
test/samples/rollup.min.js (367 KiB)
> 5.36ms

Warm Runs, All Samples (average of 25 runs)
test/samples/*.js (3635 KiB)
> 40.28ms
```

Wasm Build:

```
Module load time
> 10ms
Cold Run, All Samples
test/samples/*.js (3635 KiB)
> 43ms

Warm Runs (average of 25 runs)
test/samples/angular.js (1410 KiB)
> 9.32ms
```

```
test/samples/angular.min.js (303 KiB)
> 3.16ms
test/samples/d3.js (553 KiB)
> 5ms
test/samples/d3.min.js (250 KiB)
> 2.32ms
test/samples/magic-string.js (34 KiB)
> 0.16ms
test/samples/magic-string.min.js (20 KiB)
> 0ms
test/samples/rollup.js (698 KiB)
> 6.28ms
test/samples/rollup.min.js (367 KiB)
> 3.6ms

Warm Runs, All Samples (average of 25 runs)
test/samples/*.js (3635 KiB)
> 27.76ms
```

Wasm Build Steps

To build download the WASI SDK from https://github.com/WebAssembly/wasi-sdk/releases.

The Makefile assumes the existence of "wasi-sdk-11.0" and "wabt" (optional) as sibling folders to this project.

The build through the Makefile is then run via `make lib/lexer.wasm`, which can also be triggered via `npm run build-wasm` to create `dist/lexer.js`.

On Windows it may be preferable to use the Linux subsystem.

After the Web Assembly build, the CJS build can be triggered via `npm run build`.

Optimization passes are run with Binaryen prior to publish to reduce the Web Assembly footprint.

License

MIT