A JavaScript mangler/compressor toolkit for ES6+.

*note*: ~~You can support this project on patreon: [link]~~ **The Terser Patreon is shutting down in favor of opencollective**. Check out PATRONS.md for our first-tier patrons.

Terser recommends you use RollupJS to bundle your modules, as that produces smaller code overall.

*Beautification* has been undocumented and is *being removed* from terser, we recommend you use prettier.

Find the changelog in CHANGELOG.md

# Why choose terser?

`uglify-es` is no longer maintained and `uglify-js` does not support ES6+.

`terser` is a fork of `uglify-es` that mostly retains API and CLI compatibility with `uglify-es` and `uglify-js@3`.

# Install

First make sure you have installed the latest version of node.js (You may need to restart your computer after this step).

From NPM for use as a command line app:

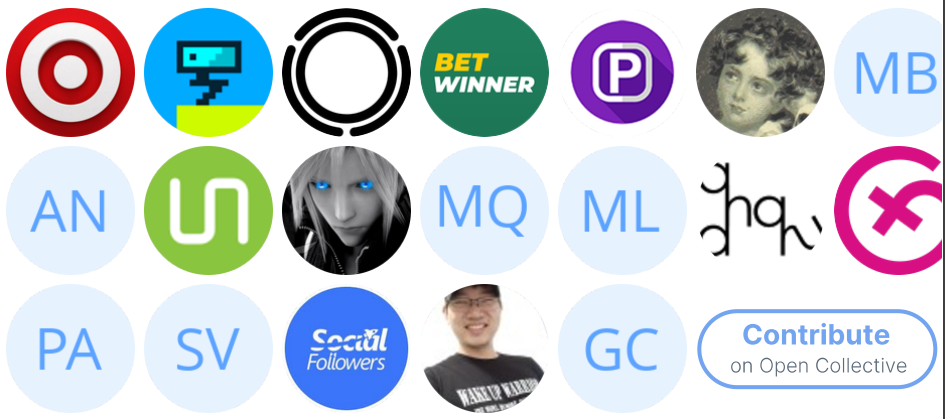
```
npm install terser -g
```

From NPM for programmatic use:

```
npm install terser
```




### Individuals




### Organizations

Support this project with your organization. Your logo will show up here with a link to your website. [Contribute]





# Command line usage

```
terser [input files] [options]
```

Terser can take multiple input files. It's recommended that you pass the input files first, then pass the options. Terser will parse input files in sequence and apply any compression options. The files are parsed in the same global scope, that is, a reference from a file to some variable/function declared in another file will be matched properly.

Command line arguments that take options (like –parse, –compress, –mangle and –format) can take in a comma-separated list of default option overrides. For instance:

```
terser input.js --compress
ecma=2015,computed_props=false
```

If no input file is specified, Terser will read from STDIN.

If you wish to pass your options before the input files, separate the two with a double dash to prevent input files being used as option arguments:

```
terser --compress --mangle -- input.js
```

## Command line options

```
    -h, --help                      Print
usage information.
```

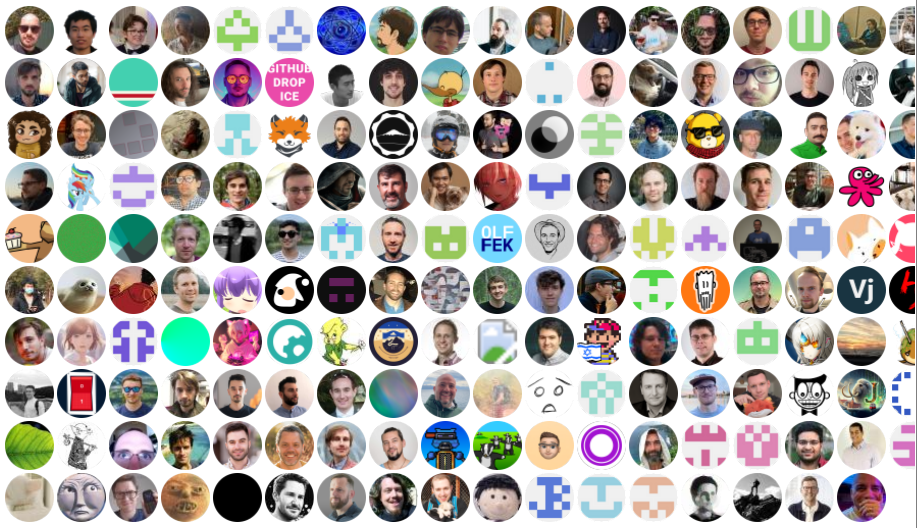
```
                                `--help
options` for details on available
options.
    -V, --version               Print
version number.
    -p, --parse <options>       Specify
parser options:
                                `acorn`
Use Acorn for parsing.

`bare_returns`  Allow return outside of
functions.

Useful when minifying CommonJS

modules and Userscripts that may

be anonymous function wrapped (IIFE)

by the .user.js engine `caller`.

`expression`  Parse a single expression,
rather than

a program (for parsing JSON).

`spidermonkey`  Assume input files are
SpiderMonkey

AST format (as JSON).
    -c, --compress [options]    Enable
compressor/specify compressor options:

`pure_funcs`  List of functions that can
be safely

removed when their return values are
```



# Contributors

## Code Contributors

This project exists thanks to all the people who contribute. [Contribute].



## Financial Contributors

Become a financial contributor and help us sustain our community. [Contribute]

# README.md Patrons:

*note*: ~~You can support this project on patreon: [link]~~ **The Terser Patreon is shutting down in favor of opencollective**. Check out PATRONS.md for our first-tier patrons.

These are the second-tier patrons. Great thanks for your support!

- CKEditor 
- 38elements



```
not used.
    -m, --mangle [options]      Mangle
names/specify mangler options:

`reserved`  List of names that should
not be mangled.
    --mangle-props [options]    Mangle
properties/specify mangler options:

`builtins`  Mangle property names that
overlaps

with standard JavaScript globals and DOM

API props.
                                `debug`
Add debug prefix and suffix.

`keep_quoted`  Only mangle unquoted
properties, quoted

properties are automatically reserved.

`strict` disables quoted properties

being automatically reserved.
                                `regex`
Only mangle matched property names.

`only_annotated` Only mangle properties
defined with /*@__MANGLE_PROP__*/.

`reserved`  List of names that should
not be mangled.
    -f, --format [options]      Specify
format options.
```

```
`preamble`  Preamble to prepend to the
output. You

can use this to insert a comment, for

example for licensing information.

This will not be parsed, but the source

map will adjust for its presence.

`quote_style`  Quote style:

0 - auto

1 - single

2 - double

3 - original

`wrap_iife`  Wrap IIFEs in parenthesis.
Note: you may

want to disable `negate_iife` under

compressor options.

`wrap_func_args`  Wrap function
arguments in parenthesis.
    -o, --output <file>         Output
file path (default STDOUT). Specify
`ast` or

`spidermonkey` to write Terser or
SpiderMonkey AST
```



If you're not sure how to set an environment variable on your shell (the above example works in bash), you can try using cross-env:

```
> npx cross-env TERSER_DEBUG_DIR=/path/
to/logs command-that-uses-terser
```

## Stack traces

In the terser CLI we use source-map-support to produce good error stacks. In your own app, you're expected to enable source-map-support (read their docs) to have nice stack traces that will help you write good issues.

# Reporting issues

## A minimal, reproducible example

You're expected to provide a [minimal reproducible example] of input code that will demonstrate your issue.

To get to this example, you can remove bits of your code and stop if your issue ceases to reproduce.

## Obtaining the source code given to Terser

Because users often don't control the call to `await minify()` or its arguments, Terser provides a `TERSER_DEBUG_DIR` environment variable to make terser output some debug logs.

These logs will contain the input code and options of each `minify()` call.

```
TERSER_DEBUG_DIR=/tmp/terser-log-dir
        command-that-uses-terser
ls /tmp/terser-log-dir
terser-debug-123456.log
```



```
                                    as JSON
to STDOUT respectively.
    --comments [filter]             Preserve
copyright comments in the output. By
                                    default
this works like Google Closure, keeping
                                    JSDoc-
style comments that contain e.g.
"@license",
                                    or start
with "!". You can optionally pass one of
the

following arguments to this flag:
                                    - "all"
to keep all comments
                                    -
`false` to omit comments in the output
                                    - a
valid JS RegExp like `/foo/` or `/^!/`
to
                                    keep
only matching comments.
                                    Note
that currently not *all* comments can be
                                    kept
when compression is on, because of dead
                                    code
removal or cascading statements into

sequences.
    --config-file <file>            Read
`minify()` options from JSON file.
    -d, --define <expr>[=value] Global
definitions.
    --ecma <version>                Specify
ECMAScript release: 5, 2015, 2016, etc.
```

```
    -e, --enclose [arg[:value]] Embed
output in a big function with
configurable

arguments and values.
    --ie8                       Support
non-standard Internet Explorer 8.

Equivalent to setting `ie8: true` in
`minify()`
                                for
`compress`, `mangle` and `format`
options.
                                By
default Terser will not try to be IE-
proof.
    --keep-classnames           Do not
mangle/drop class names.
    --keep-fnames               Do not
mangle/drop function names.  Useful for
                                code
relying on Function.prototype.name.
    --module                    Input is
an ES6 module. If `compress` or `mangle`
is
                                enabled
then the `toplevel` option, as well as
strict mode,
                                will be
enabled.
    --name-cache <file>         File to
hold mangled name mappings.
    --safari10                  Support
non-standard Safari 10/11.

Equivalent to setting `safari10: true`
in `minify()`
```



```
  "resolutions": {
    "uglify-es": "npm:terser"
  }
```

to use `terser` instead of `uglify-es` in all deeply nested dependencies without changing any code.

Note: for this change to take effect you must run the following commands to remove the existing `yarn` lock file and reinstall all packages:

```
$ rm -rf node_modules yarn.lock
$ yarn
```

- `arguments.callee`, `arguments.caller` and `Function.prototype.caller` are not used.
- The code doesn't expect the contents of `Function.prototype.toString()` or `Error.prototype.stack` to be anything in particular.
- Getting and setting properties on a plain object does not cause other side effects (using `.watch()` or `Proxy`).
- Object properties can be added, removed and modified (not prevented with `Object.defineProperty()`, `Object.defineProperties()`, `Object.freeze()`, `Object.preventExtensions()` or `Object.seal()`).
- `document.all` is not `== null`
- Assigning properties to a class doesn't have side effects and does not throw.

## Build Tools and Adaptors using Terser

https://www.npmjs.com/browse/depended/terser

## Replacing `uglify-es` with `terser` in a project using `yarn`

A number of JS bundlers and uglify wrappers are still using buggy versions of `uglify-es` and have not yet upgraded to `terser`. If you are using `yarn` you can add the following alias to your project's `package.json` file:



```
                                    for
`mangle` and `format` options.
                                    By
default `terser` will not work around
                                    Safari
10/11 bugs.
    --source-map [options]      Enable
source map/specify source map options:
                                    `base`
Path to compute relative paths from
input files.

`content`  Input source map, useful if
you're compressing

JS that was generated from some other
original

code. Specify "inline" if the source map
is

included within the sources.

`filename`  Name and/or location of the
output source.

`includeSources`  Pass this flag if you
want to include

the content of source files in the

source map as sourcesContent property.
                                    `root`
Path to the original source to be
included in

the source map.
```

```
                                  `url`
If specified, path to the source map to
append in

`//# sourceMappingURL`.
    --timings                     Display
operations run time on STDERR.
    --toplevel                    Compress
and/or mangle variables in top level
scope.
    --wrap <name>                 Embed
everything in a big function, making the

“exports” and “global” variables
available. You
                                  need to
pass an argument to this option to
                                  specify
the name that your module will take
                                  when
included in, say, a browser.
```

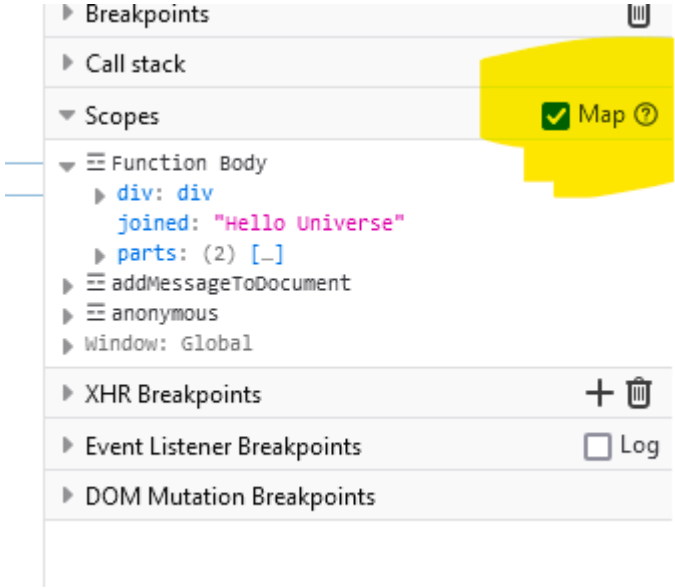
Specify `--output` (`-o`) to declare the output file. Otherwise the output goes to STDOUT.

# CLI source map options

Terser can generate a source map file, which is highly useful for debugging your compressed JavaScript. To get a source map, pass `--source-map --output output.js` (source map will be written out to `output.js.map`).



Without this, all variable values will be `undefined`. See https://github.com/terser/terser/issues/1367 for more details.




image

## Compiler assumptions

To allow for better optimizations, the compiler makes various assumptions:

- `.toString()` and `.valueOf()` don’t have side effects, and for built-in objects they have not been overridden.
- `undefined`, `NaN` and `Infinity` have not been externally redefined.

| d3.js | size | gzip size | time (s) |
|---|---|---|---|
| terser@3.7.5 mangle=true, compress=false | | | |
| terser@3.7.5 mangle=true, compress=true | 212,046 | 70,954 | 5.87 |
| babili@0.1.4 | 210,713 | 72,140 | 12.64 |
| babel-minify@0.4.3 | 210,321 | 72,242 | 48.67 |
| babel-minify@0.5.0-alpha.01eac1c3 | 210,421 | 72,238 | 14.17 |

To enable fast minify mode from the CLI use:

```
terser file.js -m
```

To enable fast minify mode with the API use:

```
await minify(code, { compress: false,
        mangle: true });
```

### Source maps and debugging

Various `compress` transforms that simplify, rearrange, inline and remove code are known to have an adverse effect on debugging with source maps. This is expected as code is optimized and mappings are often simply not possible as some code no longer exists. For highest fidelity in source map debugging disable the `compress` option and just use `mangle`.

When debugging, make sure you enable the **"map scopes"** feature to map mangled variable names back to their original names.



Additional options:

- `--source-map "filename='<NAME>'"` to specify the name of the source map.
- `--source-map "root='<URL>'"` to pass the URL where the original files can be found.
- `--source-map "url='<URL>'"` to specify the URL where the source map can be found. Otherwise Terser assumes HTTP `X-SourceMap` is being used and will omit the `//# sourceMappingURL=` directive.

For example:

```
terser js/file1.js js/file2.js \
        -o foo.min.js -c -m \
        --source-map "root='http://
foo.com/src',url='foo.min.js.map'"
```

The above will compress and mangle `file1.js` and `file2.js`, will drop the output in `foo.min.js` and the source map in `foo.min.js.map`. The source mapping will refer to `http://foo.com/src/js/file1.js` and `http://foo.com/src/js/file2.js` (in fact it will list `http://foo.com/src` as the source map root, and the original files as `js/file1.js` and `js/file2.js`).

## Composed source map

When you're compressing JS code that was output by a compiler such as CoffeeScript, mapping to the JS code won't be too helpful. Instead, you'd like to map back to the original code

(i.e. CoffeeScript). Terser has an option to take an input source map. Assuming you have a mapping from CoffeeScript → compiled JS, Terser can generate a map from CoffeeScript → compressed JS by mapping every token in the compiled JS to its original location.

To use this feature pass `--source-map "content='/path/to/input/source.map'"` or `--source-map "content=inline"` if the source map is included inline with the sources.

# CLI compress options

You need to pass `--compress` (`-c`) to enable the compressor. Optionally you can pass a comma-separated list of compress options.

Options are in the form `foo=bar`, or just `foo` (the latter implies a boolean option that you want to set `true`; it's effectively a shortcut for `foo=true`).

Example:

```
terser file.js -c
toplevel,sequences=false
```



`spidermonkey` is also available in `minify` as `parse` and `format` options to accept and/or produce a spidermonkey AST.

## Use Acorn for parsing

More for fun, I added the `-p acorn` option which will use Acorn to do all the parsing. If you pass this option, Terser will `require("acorn")`.

Acorn is really fast (e.g. 250ms instead of 380ms on some 650K code), but converting the SpiderMonkey tree that Acorn produces takes another 150ms so in total it's a bit more than just using Terser's own parser.

## Terser Fast Minify Mode

It's not well known, but whitespace removal and symbol mangling accounts for 95% of the size reduction in minified code for most JavaScript - not elaborate code transforms. One can simply disable `compress` to speed up Terser builds by 3 to 4 times.

| **d3.js** | **size** | **gzip size** | **time (s)** |
|---|---|---|---|
| original | 451,131 | 108,733 | - |
| terser@3.7.5 mangle=false, compress=false | 316,600 | 85,245 | 0.82 |
| | 220,216 | 72,730 | 1.45 |

```
/*#__NOINLINE__*/
function_cant_be_inlined_into_here()

const x = /*#__PURE__*/
        i_am_dropped_if_x_is_not_used()

function lookup(object, key) { return
        object[key]; }
lookup({ i_will_be_mangled_too:
        "bar" }, /*@__KEY__*/
        "i_will_be_mangled_too");
```

### ESTree / SpiderMonkey AST

Terser has its own abstract syntax tree format; for practical reasons we can't easily change to using the SpiderMonkey AST internally. However, Terser now has a converter which can import a SpiderMonkey AST.

For example Acorn is a super-fast parser that produces a SpiderMonkey AST. It has a small CLI utility that parses one file and dumps the AST in JSON on the standard output. To use Terser to mangle and compress that:

```
acorn file.js | terser -p spidermonkey
-m -c
```

The `-p spidermonkey` option tells Terser that all input files are not JavaScript, but JS code described in SpiderMonkey AST in JSON. Therefore we don't use our own parser in this case, but just transform that AST into our internal AST.



## CLI mangle options

To enable the mangler you need to pass `--mangle` (`-m`). The following (comma-separated) options are supported:

- `toplevel` (default `false`) – mangle names declared in the top level scope.
- `eval` (default `false`) – mangle names visible in scopes where `eval` or `with` are used.

When mangling is enabled but you want to prevent certain names from being mangled, you can declare those names with `--mangle reserved` — pass a comma-separated list of names. For example:

```
terser ... -m
reserved=['$','require','exports']
```

to prevent the `require`, `exports` and `$` names from being changed.

### CLI mangling property names (`--mangle-props`)

**Note:** THIS **WILL** BREAK YOUR CODE. A good rule of thumb is not to use this unless you know exactly what you're doing and how this works and read this section until the end.

Mangling property names is a separate step, different from variable name mangling. Pass `--mangle-props` to enable it.

The least dangerous way to use this is to use the `regex` option like so:

```
terser example.js -c -m --mangle-props
regex=/_$/
```

This will mangle all properties that end with an underscore. So you can use it to mangle internal methods.

By default, it will mangle all properties in the input code with the exception of built in DOM properties and properties in core JavaScript classes, which is what will break your code if you don't:

1. Control all the code you're mangling
2. Avoid using a module bundler, as they usually will call Terser on each file individually, making it impossible to pass mangled objects between modules.
3. Avoid calling functions like `defineProperty` or `hasOwnProperty`, because they refer to object properties using strings and will break your code if you don't know what you are doing.

An example:

```
// example.js
var x = {
    baz_: 0,
    foo_: 1,
    calc: function() {
        return this.foo_ + this.baz_;
    }
};
x.bar_ = 2;
```



```
await minify("alert('hello');", {
    compress: {
        global_defs: {
            "alert": "console.log"
        }
    }
}).code;
// returns: '"console.log"("hello");'
```

## Annotations

Annotations in Terser are a way to tell it to treat a certain function call differently. The following annotations are available:

- `/*@__INLINE__*/` - forces a function to be inlined somewhere.
- `/*@__NOINLINE__*/` - Makes sure the called function is not inlined into the call site.
- `/*@__PURE__*/` - Marks a function call as pure. That means, it can safely be dropped.
- `/*@__KEY__*/` - Marks a string literal as a property to also mangle it when mangling properties.
- `/*@__MANGLE_PROP__*/` - Opts-in an object property (or class field) for mangling, when the property mangler is enabled.

You can use either a @ sign at the start, or a #.

Here are some examples on how to use them:

```
/*@__INLINE__*/
function_always_inlined_here()
```

## Conditional compilation API

You can also use conditional compilation via the programmatic API. With the difference that the property name is `global_defs` and is a compressor property:

```
var result = await
        minify(fs.readFileSync("input.js",
        "utf8"), {
    compress: {
        dead_code: true,
        global_defs: {
            DEBUG: false
        }
    }
});
```

To replace an identifier with an arbitrary non-constant expression it is necessary to prefix the `global_defs` key with `"@"` to instruct Terser to parse the value as an expression:

```
await minify("alert('hello');", {
    compress: {
        global_defs: {
            "@alert": "console.log"
        }
    }
}).code;
// returns: 'console.log("hello");'
```

Otherwise it would be replaced as string literal:



```
x["baz_"] = 3;
console.log(x.calc());
```

Mangle all properties (except for JavaScript `builtins`) (**very** unsafe):

```
$ terser example.js -c passes=2 -m --
        mangle-props

var x={o:3,t:1,i:function(){return
        this.t+this.o},s:2};console.log(x.i());
```

Mangle all properties except for `reserved` properties (still very unsafe):

```
$ terser example.js -c passes=2 -m --
        mangle-props
        reserved=[foo_,bar_]

var x={o:3,foo_:1,t:function(){return
        this.foo_+this.o},bar_:2};console.log(x
```

Mangle all properties matching a `regex` (not as unsafe but still unsafe):

```
$ terser example.js -c passes=2 -m --
        mangle-props regex=/_$/

var x={o:3,t:1,calc:function(){return
        this.t+this.o},i:2};console.log(x.calc(
```

Combining mangle properties options:

```
$ terser example.js -c passes=2 -m --
        mangle-props regex=/
        _$/,reserved=[bar_]

var x={o:3,t:1,calc:function(){return
        this.t+this.o},bar_:2};console.log(x.calc()
```

In order for this to be of any use, we avoid mangling standard JS names and DOM API properties by default (`--mangle-props builtins` to override).

A regular expression can be used to define which property names should be mangled. For example, `--mangle-props regex=/^_/` will only mangle property names that start with an underscore.

When you compress multiple files using this option, in order for them to work together in the end we need to ensure somehow that one property gets mangled to the same name in all of them. For this, pass `--name-cache filename.json` and Terser will maintain these mappings in a file which can then be reused. It should be initially empty. Example:

```
$ rm -f /tmp/cache.json  # start fresh
$ terser file1.js file2.js --mangle-
        props --name-cache /tmp/
        cache.json -o part1.js
$ terser file3.js file4.js --mangle-
        props --name-cache /tmp/
        cache.json -o part2.js
```

Now, `part1.js` and `part2.js` will be consistent with each other in terms of mangled property names.



`DEBUG=false` then, coupled with dead code removal Terser will discard the following from the output:

```
if (DEBUG) {
    console.log("debug stuff");
}
```

You can specify nested constants in the form of `--define env.DEBUG=false`.

Another way of doing that is to declare your globals as constants in a separate file and include it into the build. For example you can have a `build/defines.js` file with the following:

```
var DEBUG = false;
var PRODUCTION = true;
// etc.
```

and build your code like this:

```
terser build/defines.js js/foo.js js/
bar.js... -c
```

Terser will notice the constants and, since they cannot be altered, it will evaluate references to them to the value itself and drop unreachable code as usual. The build will contain the `const` declarations if you use them. If you are targeting < ES6 environments which does not support `const`, using `var` with `reduce_vars` (enabled by default) should suffice.

The safest comments where to place copyright information (or other info that needs to be kept in the output) are comments attached to toplevel nodes.

## The `unsafe compress` option

It enables some transformations that *might* break code logic in certain contrived cases, but should be fine for most code. It assumes that standard built-in ECMAScript functions and classes have not been altered or replaced. You might want to try it on your own code; it should reduce the minified size. Some examples of the optimizations made when this option is enabled:

- `new Array(1, 2, 3)` or `Array(1, 2, 3)` → `[ 1, 2, 3 ]`
- `Array.from([1, 2, 3])` → `[1, 2, 3]`
- `new Object()` → `{}`
- `String(exp)` or `exp.toString()` → `"" + exp`
- `new Object/RegExp/Function/Error/Array (...)` → we discard the `new`
- `"foo bar".substr(4)` → `"bar"`

## Conditional compilation

You can use the `--define` (`-d`) switch in order to declare global variables that Terser will assume to be constants (unless defined in scope). For example if you pass `--define`



Using the name cache is not necessary if you compress all your files in a single call to Terser.

## Mangling unquoted names (`--mangle-props keep_quoted`)

Using quoted property name (`o["foo"]`) reserves the property name (`foo`) so that it is not mangled throughout the entire script even when used in an unquoted style (`o.foo`). Example:

```
// stuff.js
var o = {
    "foo": 1,
    bar: 3
};
o.foo += o.bar;
console.log(o.foo);

$ terser stuff.js --mangle-props
        keep_quoted -c -m

var
        o={foo:1,o:3};o.foo+=o.o,console.log(o.
```

## Debugging property name mangling

You can also pass `--mangle-props debug` in order to mangle property names without completely obscuring them. For

example the property `o.foo` would mangle to `o._$foo$_` with this option. This allows property mangling of a large codebase while still being able to debug the code and identify where mangling is breaking things.

```
$ terser stuff.js --mangle-props debug
        -c -m

var
        o={_$foo$_:1,_$bar$_:3};o._$foo$_+=o._$bar$
```

You can also pass a custom suffix using `--mangle-props debug=XYZ`. This would then mangle `o.foo` to `o._$foo$XYZ_`. You can change this each time you compile a script to identify how a property got mangled. One technique is to pass a random number on every compile to simulate mangling changing with different inputs (e.g. as you update the input script with new properties), and to help identify mistakes like writing mangled keys to storage.



# Miscellaneous

## Keeping copyright notices or other comments

You can pass `--comments` to retain certain comments in the output. By default it will keep comments starting with "!" and JSDoc-style comments that contain "@preserve", "@copyright", "@license" or "@cc_on" (conditional compilation for IE). You can pass `--comments all` to keep all the comments, or a valid JavaScript regexp to keep only comments that match this regexp. For example `--comments /^!/` will keep comments like `/*! Copyright Notice */`.

Note, however, that there might be situations where comments are lost. For example:

```
function f() {
    /** @preserve Foo Bar */
    function g() {
        // this function is never called
    }
    return something();
}
```

Even though it has "@preserve", the comment will be lost because the inner function `g` (which is the AST node to which the comment is attached to) is discarded by the compressor as not referenced.

- `semicolons` (default `true`) – separate statements with semicolons. If you pass `false` then whenever possible we will use a newline instead of a semicolon, leading to more readable output of minified code (size before gzip could be smaller; size after gzip insignificantly larger).
- `shebang` (default `true`) – preserve shebang `#!` in preamble (bash scripts)
- `spidermonkey` (default `false`) – produce a Spidermonkey (Mozilla) AST
- `webkit` (default `false`) – enable workarounds for WebKit bugs. PhantomJS users should set this option to `true`.
- `wrap_iife` (default `false`) – pass `true` to wrap immediately invoked function expressions. See #640 for more details.
- `wrap_func_args` (default `false`) – pass `true` in order to wrap function expressions that are passed as arguments, in parenthesis. See OptimizeJS for more details.



# API Reference

Assuming installation via NPM, you can load Terser in your application like this:

```
const { minify } = require("terser");
```

Or,

```
import { minify } from "terser";
```

Browser loading is also supported. It exposes a global variable `Terser` containing a `.minify` property:

```
<script src="https://cdn.jsdelivr.net/
        npm/source-map@0.7.3/dist/
        source-map.js"></script>
<script src="https://cdn.jsdelivr.net/
        npm/terser/dist/
        bundle.min.js"></script>
```

There is an async high level function, `async minify(code, options)`, which will perform all minification phases in a configurable manner. By default `minify()` will enable `compress` and `mangle`. Example:

```
var code = "function add(first, second)
        { return first + second; }";
var result = await minify(code, {
        sourceMap: true });
```

```
console.log(result.code);  // minified
        output: function add(n,d){return
        n+d}
console.log(result.map);  // source map
```

There is also a `minify_sync()` alternative version of it, which returns instantly.

You can `minify` more than one JavaScript file at a time by using an object for the first argument where the keys are file names and the values are source code:

```
var code = {
    "file1.js": "function add(first,
        second) { return first +
        second; }",
    "file2.js": "console.log(add(1 + 2,
        3 + 4));"
};
var result = await minify(code);
console.log(result.code);
// function add(d,n){return d+n}
        console.log(add(3,7));
```

The `toplevel` option:

```
var code = {
    "file1.js": "function add(first,
        second) { return first +
        second; }",
    "file2.js": "console.log(add(1 + 2,
        3 + 4));"
};
var options = { toplevel: true };
```



- `inline_script` (default `true`) – escape HTML comments and the slash in occurrences of `</script>` in strings
- `keep_numbers` (default `false`) – keep number literals as it was in original code (disables optimizations like converting `1000000` into `1e6`)
- `keep_quoted_props` (default `false`) – when turned on, prevents stripping quotes from property names in object literals.
- `max_line_len` (default `false`) – maximum line length (for minified code)
- `preamble` (default `null`) – when passed it must be a string and it will be prepended to the output literally. The source map will adjust for this text. Can be used to insert a comment containing licensing information, for example.
- `quote_keys` (default `false`) – pass `true` to quote all keys in literal objects
- `quote_style` (default `0`) – preferred quote style for strings (affects quoted property names and directives as well):
  - `0` – prefers double quotes, switches to single quotes when there are more double quotes in the string itself. `0` is best for gzip size.
  - `1` – always use single quotes
  - `2` – always use double quotes
  - `3` – always use the original quotes
- `preserve_annotations` – (default `false`) – Preserve Terser annotations in the output.
- `safari10` (default `false`) – set this option to `true` to work around the Safari 10/11 await bug. See also: the `safari10` mangle option.

# Format options

These options control the format of Terser's output code. Previously known as "output options".

- `ascii_only` (default `false`) – escape Unicode characters in strings and regexps (affects directives with non-ascii characters becoming invalid)
- `beautify` (default `false`) – (DEPRECATED) whether to beautify the output. When using the legacy `-b` CLI flag, this is set to true by default.
- `braces` (default `false`) – always insert braces in `if`, `for`, `do`, `while` or `with` statements, even if their body is a single statement.
- `comments` (default `"some"`) – by default it keeps JSDoc-style comments that contain "@license", "@copyright", "@preserve" or start with `!`, pass `true` or `"all"` to preserve all comments, `false` to omit comments in the output, a regular expression string (e.g. `/^!/`) or a function.
- `ecma` (default `5`) – set desired EcmaScript standard version for output. Set `ecma` to `2015` or greater to emit shorthand object properties - i.e.: `{a}` instead of `{a: a}`. The `ecma` option will only change the output in direct control of the beautifier. Non-compatible features in your input will still be output as is. For example: an `ecma` setting of `5` will **not** convert modern code to ES5.
- `indent_level` (default `4`)
- `indent_start` (default `0`) – prefix all lines by that many spaces



```
var result = await minify(code,
        options);
console.log(result.code);
// console.log(3+7);
```

The `nameCache` option:

```
var options = {
    mangle: {
        toplevel: true,
    },
    nameCache: {}
};
var result1 = await minify({
    "file1.js": "function add(first,
        second) { return first +
        second; }"
}, options);
var result2 = await minify({
    "file2.js": "console.log(add(1 + 2,
        3 + 4));"
}, options);
console.log(result1.code);
// function n(n,r){return n+r}
console.log(result2.code);
// console.log(n(3,7));
```

You may persist the name cache to the file system in the following way:

```
var cacheFileName = "/tmp/cache.json";
var options = {
    mangle: {
        properties: true,
```

```
    },
    nameCache:
        JSON.parse(fs.readFileSync(cacheFileName,
        "utf8"))
};
fs.writeFileSync("part1.js", await
        minify({
    "file1.js":
        fs.readFileSync("file1.js",
        "utf8"),
    "file2.js":
        fs.readFileSync("file2.js",
        "utf8")
}, options).code, "utf8");
fs.writeFileSync("part2.js", await
        minify({
    "file3.js":
        fs.readFileSync("file3.js",
        "utf8"),
    "file4.js":
        fs.readFileSync("file4.js",
        "utf8")
}, options).code, "utf8");
fs.writeFileSync(cacheFileName,
        JSON.stringify(options.nameCache),
        "utf8");
```

An example of a combination of `minify()` options:

```
var code = {
    "file1.js": "function add(first,
        second) { return first +
        second; }",
```



- `keep_quoted` (default: `false`) — How quoting properties (`{"prop": ...}` and `obj["prop"]`) controls what gets mangled.
  - `"strict"` (recommended) – `obj.prop` is mangled.
  - `false` – `obj["prop"]` is mangled.
  - `true` – `obj.prop` is mangled unless there is `obj["prop"]` elsewhere in the code.
- `nth_identifier` (default: an internal mangler that weights based on character frequency analysis) – Pass an object with a `get(n)` function that converts an ordinal into the nth most favored (usually shortest) identifier. Optionally also provide `reset()`, `sort()`, and `consider(chars, delta)` to use character frequency analysis of the source code.
- `regex` (default: `null`) — Pass a RegExp literal or pattern string to only mangle property matching the regular expression.
- `reserved` (default: `[]`) — Do not mangle property names listed in the `reserved` array.
- `undeclared` (default: `false`) - Mangle those names when they are accessed as properties of known top level variables but their declarations are never found in input code. May be useful when only minifying parts of a project. See #397 for more details.

```
function funcName(firstLongName,
        anotherLongName) {
    var myVariable = firstLongName +
        anotherLongName;
}

var code = fs.readFileSync("test.js",
        "utf8");

await minify(code).code;
// 'function funcName(a,n){}var
        globalVar;'

await minify(code, { mangle: {
        reserved:
        ['firstLongName'] } }).code;
// 'function funcName(firstLongName,a){}
        var globalVar;'

await minify(code, { mangle: {
        toplevel: true } }).code;
// 'function n(n,a){}var a;'
```

## Mangle properties options

- `builtins` (default: `false`) — Use `true` to allow the mangling of builtin DOM properties. Not recommended to override this setting.
- `debug` (default: `false`) — Mangle names with the original name still present. Pass an empty string `""` to enable, or a non-empty string to set the debug suffix.



```
    "file2.js": "console.log(add(1 + 2,
        3 + 4));"
};
var options = {
    toplevel: true,
    compress: {
        global_defs: {
            "@console.log": "alert"
        },
        passes: 2
    },
    format: {
        preamble: "/* minified */"
    }
};
var result = await minify(code,
        options);
console.log(result.code);
// /* minified */
// alert(10);"
```

An error example:

```
try {
    const result = await
        minify({"foo.js" : "if (0) else
        console.log(1);"});
    // Do something with result
} catch (error) {
    const { message, filename, line,
        col, pos } = error;
    // Do something with error
}
```

# Minify options

- `ecma` (default `undefined`) - pass `5`, `2015`, `2016`, etc to override `compress` and `format`'s `ecma` options.
- `enclose` (default `false`) - pass `true`, or a string in the format of `"args[:values]"`, where `args` and `values` are comma-separated argument names and values, respectively, to embed the output in a big function with the configurable arguments and values.
- `parse` (default `{}`) — pass an object if you wish to specify some additional parse options.
- `compress` (default `{}`) — pass `false` to skip compressing entirely. Pass an object to specify custom compress options.
- `mangle` (default `true`) — pass `false` to skip mangling names, or pass an object to specify mangle options (see below).
  - `mangle.properties` (default `false`) — a subcategory of the mangle option. Pass an object to specify custom mangle property options.
- `module` (default `false`) — Use when minifying an ES6 module. "use strict" is implied and names can be mangled on the top scope. If `compress` or `mangle` is enabled then the `toplevel` option will be enabled.
- `format` or `output` (default `null`) — pass an object if you wish to specify additional format options. The defaults are optimized for best compression.
- `sourceMap` (default `false`) - pass an object if you wish to specify source map options.



names matching that regex. See also: the `keep_classnames` compress option.

- `keep_fnames` (default `false`) – Pass `true` to not mangle function names. Pass a regular expression to only keep function names matching that regex. Useful for code relying on `Function.prototype.name`. See also: the `keep_fnames` compress option.
- `module` (default `false`) – Pass `true` an ES6 modules, where the toplevel scope is not the global scope. Implies `toplevel` and assumes input code is strict mode JS.
- `nth_identifier` (default: an internal mangler that weights based on character frequency analysis) – Pass an object with a `get(n)` function that converts an ordinal into the nth most favored (usually shortest) identifier. Optionally also provide `reset()`, `sort()`, and `consider(chars, delta)` to use character frequency analysis of the source code.
- `reserved` (default `[]`) – Pass an array of identifiers that should be excluded from mangling. Example: `["foo", "bar"]`.
- `toplevel` (default `false`) – Pass `true` to mangle names declared in the top level scope.
- `safari10` (default `false`) – Pass `true` to work around the Safari 10 loop iterator bug "Cannot declare a let variable twice". See also: the `safari10` format option.

  Examples:

```
// test.js
var globalVar;
```

- `unsafe_methods` (default: false) – Converts `{ m: function(){} }` to `{ m(){} }`. `ecma` must be set to `6` or greater to enable this transform. If `unsafe_methods` is a RegExp then key/value pairs with keys matching the RegExp will be converted to concise methods. Note: if enabled there is a risk of getting a "`<method name>` is not a constructor" TypeError should any code try to `new` the former function.
- `unsafe_proto` (default: `false`) – optimize expressions like `Array.prototype.slice.call(a)` into `[].slice.call(a)`
- `unsafe_regexp` (default: `false`) – enable substitutions of variables with `RegExp` values the same way as if they are constants.
- `unsafe_undefined` (default: `false`) – substitute `void 0` if there is a variable named `undefined` in scope (variable name will be mangled, typically reduced to a single character)
- `unused` (default: `true`) – drop unreferenced functions and variables (simple direct variable assignments do not count as references unless set to `"keep_assign"`)

# Mangle options

- `eval` (default `false`) – Pass `true` to mangle names visible in scopes where `eval` or `with` are used.
- `keep_classnames` (default `false`) – Pass `true` to not mangle class names. Pass a regular expression to only keep class



- `toplevel` (default `false`) - set to `true` if you wish to enable top level variable and function name mangling and to drop unused variables and functions.
- `nameCache` (default `null`) - pass an empty object `{}` or a previously used `nameCache` object if you wish to cache mangled variable and property names across multiple invocations of `minify()`. Note: this is a read/write property. `minify()` will read the name cache state of this object and update it during minification so that it may be reused or externally persisted by the user.
- `ie8` (default `false`) - set to `true` to support IE8.
- `keep_classnames` (default: `undefined`) - pass `true` to prevent discarding or mangling of class names. Pass a regular expression to only keep class names matching that regex.
- `keep_fnames` (default: `false`) - pass `true` to prevent discarding or mangling of function names. Pass a regular expression to only keep function names matching that regex. Useful for code relying on `Function.prototype.name`. If the top level minify option `keep_classnames` is `undefined` it will be overridden with the value of the top level minify option `keep_fnames`.
- `safari10` (default: `false`) - pass `true` to work around Safari 10/11 bugs in loop scoping and `await`. See `safari10` options in `mangle` and `format` for details.

## Minify options structure

```
{
    parse: {
        // parse options
    },
    compress: {
        // compress options
    },
    mangle: {
        // mangle options

        properties: {
            // mangle property options
        }
    },
    format: {
        // format options (can also use
        `output` for backwards
        compatibility)
    },
    sourceMap: {
        // source map options
    },
    ecma:
        5, // specify one of: 5, 2015,
        2016, etc.
    enclose: false, // or specify true,
        or "args:values"
    keep_classnames: false,
    keep_fnames: false,
    ie8: false,
```



set this value to `false` for IE10 and earlier versions due to known issues.

- `unsafe` (default: `false`) – apply "unsafe" transformations (details).
- `unsafe_arrows` (default: `false`) – Convert ES5 style anonymous function expressions to arrow functions if the function body does not reference `this`. Note: it is not always safe to perform this conversion if code relies on the the function having a `prototype`, which arrow functions lack. This transform requires that the `ecma` compress option is set to `2015` or greater.
- `unsafe_comps` (default: `false`) – Reverse `<` and `<=` to `>` and `>=` to allow improved compression. This might be unsafe when an at least one of two operands is an object with computed values due the use of methods like `get`, or `valueOf`. This could cause change in execution order after operands in the comparison are switching. Or if one of two operands is `NaN`, the result is always `false`. Compression only works if both `comparisons` and `unsafe_comps` are both set to true.
- `unsafe_Function` (default: `false`) – compress and mangle `Function(args, code)` when both `args` and `code` are string literals.
- `unsafe_math` (default: `false`) – optimize numerical expressions like `2 * x * 3` into `6 * x`, which may give imprecise floating point results.
- `unsafe_symbols` (default: `false`) – removes keys from native Symbol declarations, e.g `Symbol("kDog")` becomes `Symbol()`.

- `reduce_funcs` (default: `true`) – Inline single-use functions when possible. Depends on `reduce_vars` being enabled. Disabling this option sometimes improves performance of the output code.
- `sequences` (default: `true`) – join consecutive simple statements using the comma operator. May be set to a positive integer to specify the maximum number of consecutive comma sequences that will be generated. If this option is set to `true` then the default `sequences` limit is `200`. Set option to `false` or `0` to disable. The smallest `sequences` length is `2`. A `sequences` value of `1` is grandfathered to be equivalent to `true` and as such means `200`. On rare occasions the default sequences limit leads to very slow compress times in which case a value of `20` or less is recommended.
- `side_effects` (default: `true`) – Remove expressions which have no side effects and whose results aren't used.
- `switches` (default: `true`) – de-duplicate and remove unreachable `switch` branches
- `toplevel` (default: `false`) – drop unreferenced functions (`"funcs"`) and/or variables (`"vars"`) in the top level scope (`false` by default, `true` to drop both unreferenced functions and variables)
- `top_retain` (default: `null`) – prevent specific toplevel functions and variables from `unused` removal (can be array, comma-separated, RegExp or function. Implies `toplevel`)
- `typeofs` (default: `true`) – Transforms `typeof foo == "undefined"` into `foo === void 0`. Note: recommend to



```
    module: false,
    nameCache: null, // or specify a
        name cache object
    safari10: false,
    toplevel: false
}
```

## Source map options

To generate a source map:

```
var result = await minify({"file1.js":
        "var a = function() {};"}, {
    sourceMap: {
        filename: "out.js",
        url: "out.js.map"
    }
});
console.log(result.code); // minified
        output
console.log(result.map);  // source map
```

Note that the source map is not saved in a file, it's just returned in `result.map`. The value passed for `sourceMap.url` is only used to set `//# sourceMappingURL=out.js.map` in `result.code`. The value of `filename` is only used to set `file` attribute (see the spec) in source map file.

You can set option `sourceMap.url` to be `"inline"` and source map will be appended to code.

You can also specify sourceRoot property to be included in source map:

```
var result = await minify({"file1.js":
        "var a = function() {};"}, {
    sourceMap: {
        root: "http://example.com/src",
        url: "out.js.map"
    }
});
```

If you're compressing compiled JavaScript and have a source map for it, you can use `sourceMap.content`:

```
var result = await
        minify({"compiled.js":
        "compiled code"}, {
    sourceMap: {
        content: "content from
        compiled.js.map",
        url: "minified.js.map"
    }
});
// same as before, it returns `code` and
        `map`
```

If you're using the `X-SourceMap` header instead, you can just omit `sourceMap.url`.

If you happen to need the source map as a raw object, set `sourceMap.asObject` to `true`.



- `negate_iife` (default: `true`) – negate "Immediately-Called Function Expressions" where the return value is discarded, to avoid the parens that the code generator would insert.
- `passes` (default: `1`) – The maximum number of times to run compress. In some cases more than one pass leads to further compressed code. Keep in mind more passes will take more time.
- `properties` (default: `true`) – rewrite property access using the dot notation, for example `foo["bar"] → foo.bar`
- `pure_funcs` (default: `null`) – You can pass an array of names and Terser will assume that those functions do not produce side effects. DANGER: will not check if the name is redefined in scope. An example case here, for instance `var q = Math.floor(a/b)`. If variable `q` is not used elsewhere, Terser will drop it, but will still keep the `Math.floor(a/b)`, not knowing what it does. You can pass `pure_funcs: [ 'Math.floor' ]` to let it know that this function won't produce any side effect, in which case the whole statement would get discarded. The current implementation adds some overhead (compression will be slower).
- `pure_getters` (default: `"strict"`) – If you pass `true` for this, Terser will assume that object property access (e.g. `foo.bar` or `foo["bar"]`) doesn't have any side effects. Specify `"strict"` to treat `foo.bar` as side-effect-free only when `foo` is certain to not throw, i.e. not `null` or `undefined`.
- `pure_new` (default: `false`) – Set to `true` to assume `new X()` never has side effects.
- `reduce_vars` (default: `true`) – Improve optimization on variables assigned with and used as constant values.

- 3 – inline functions with arguments and variables
- `true` – same as 3

- `join_vars` (default: `true`) – join consecutive `var`, `let` and `const` statements
- `keep_classnames` (default: `false`) – Pass `true` to prevent the compressor from discarding class names. Pass a regular expression to only keep class names matching that regex. See also: the `keep_classnames` mangle option.
- `keep_fargs` (default: `true`) – Prevents the compressor from discarding unused function arguments. You need this for code which relies on `Function.length`.
- `keep_fnames` (default: `false`) – Pass `true` to prevent the compressor from discarding function names. Pass a regular expression to only keep function names matching that regex. Useful for code relying on `Function.prototype.name`. See also: the `keep_fnames` mangle option.
- `keep_infinity` (default: `false`) – Pass `true` to prevent `Infinity` from being compressed into `1/0`, which may cause performance issues on Chrome.
- `lhs_constants` (default: `true`) – Moves constant values to the left-hand side of binary nodes. `foo == 42` → `42 == foo`
- `loops` (default: `true`) – optimizations for `do`, `while` and `for` loops when we can statically determine the condition.
- `module` (default `false`) – Pass `true` when compressing an ES6 module. Strict mode is implied and the `toplevel` option as well.



# Parse options

- `bare_returns` (default `false`) – support top level `return` statements
- `html5_comments` (default `true`)
- `shebang` (default `true`) – support `#!command` as the first line
- `spidermonkey` (default `false`) – accept a Spidermonkey (Mozilla) AST

# Compress options

- `defaults` (default: `true`) – Pass `false` to disable most default enabled `compress` transforms. Useful when you only want to enable a few `compress` options while disabling the rest.
- `arrows` (default: `true`) – Class and object literal methods are converted will also be converted to arrow expressions if the resultant code is shorter: `m(){return x}` becomes `m:()=>x`. To do this to regular ES5 functions which don't use `this` or `arguments`, see `unsafe_arrows`.
- `arguments` (default: `false`) – replace `arguments[index]` with function parameter name whenever possible.
- `booleans` (default: `true`) – various optimizations for boolean context, for example `!!a ? b : c` → `a ? b : c`

- `booleans_as_integers` (default: `false`) – Turn booleans into 0 and 1, also makes comparisons with booleans use `==` and `!=` instead of `===` and `!==`.
- `collapse_vars` (default: `true`) – Collapse single-use non-constant variables, side effects permitting.
- `comparisons` (default: `true`) – apply certain optimizations to binary nodes, e.g. `!(a <= b)` → `a > b` (only when `unsafe_comps`), attempts to negate binary nodes, e.g. `a = !b && !c && !d && !e` → `a=!(b||c||d||e)` etc. Note: `comparisons` works best with `lhs_constants` enabled.
- `computed_props` (default: `true`) – Transforms constant computed properties into regular ones: `{["computed"]: 1}` is converted to `{computed: 1}`.
- `conditionals` (default: `true`) – apply optimizations for `if`-s and conditional expressions
- `dead_code` (default: `true`) – remove unreachable code
- `directives` (default: `true`) – remove redundant or non-standard directives
- `drop_console` (default: `false`) – Pass `true` to discard calls to `console.*` functions. If you only want to discard a portion of console, you can pass an array like this `['log', 'info']`, which will only discard `console.log`、`console.info`.
- `drop_debugger` (default: `true`) – remove `debugger;` statements



- `ecma` (default: `5`) – Pass `2015` or greater to enable `compress` options that will transform ES5 code into smaller ES6+ equivalent forms.
- `evaluate` (default: `true`) – attempt to evaluate constant expressions
- `expression` (default: `false`) – Pass `true` to preserve completion values from terminal statements without `return`, e.g. in bookmarklets.
- `global_defs` (default: `{}`) – see conditional compilation
- `hoist_funs` (default: `false`) – hoist function declarations
- `hoist_props` (default: `true`) – hoist properties from constant object and array literals into regular variables subject to a set of constraints. For example: `var o={p:1, q:2}; f(o.p, o.q);` is converted to `f(1, 2);`. Note: `hoist_props` works best with `mangle` enabled, the `compress` option `passes` set to `2` or higher, and the `compress` option `toplevel` enabled.
- `hoist_vars` (default: `false`) – hoist `var` declarations (this is `false` by default because it seems to increase the size of the output in general)
- `if_return` (default: `true`) – optimizations for if/return and if/continue
- `inline` (default: `true`) – inline calls to function with simple/`return` statement:
  - `false` – same as `0`
  - `0` – disabled inlining
  - `1` – inline simple functions
  - `2` – inline functions with arguments