# Acorn

A tiny, fast JavaScript parser written in JavaScript.

## Community

Acorn is open source software released under an MIT license.

You are welcome to report bugs or create pull requests on github.

## Installation

The easiest way to install acorn is from `npm`:

```
npm install acorn
```

Alternately, you can download the source and build acorn yourself:

```
git clone https://github.com/acornjs/
        acorn.git
cd acorn
npm install
```

# Interface

**parse**`(input, options)` is the main interface to the library. The `input` parameter is a string, `options` must be an object setting some of the options listed below. The return value will be an abstract syntax tree object as specified by the ESTree spec.

```
let acorn = require("acorn");
console.log(acorn.parse("1 + 1",
        {ecmaVersion: 2020}));
```

When encountering a syntax error, the parser will raise a `SyntaxError` object with a meaningful message. The error object will have a `pos` property that indicates the string offset at which the error occurred, and a `loc` object that contains a `{line, column}` object referring to that same position.

Options are provided by in a second argument, which should be an object containing any of these fields (only `ecmaVersion` is required):

- **ecmaVersion**: Indicates the ECMAScript version to parse. Can be a number, either in year (`2022`) or plain version number (`6`) form, or `"latest"` (the latest the library supports). This influences support for strict mode, the set of reserved words, and support for new syntax features.

  **NOTE**: Only ‘stage 4’ (finalized) ECMAScript features are being implemented by Acorn. Other proposed new features must be implemented through plugins.

- **sourceType**: Indicate the mode the code should be parsed in. Can be either `"script"` or `"module"`. This influences global strict mode and parsing of `import` and `export` declarations.

  **NOTE**: If set to `"module"`, then static `import` / `export` syntax will be valid, even if `ecmaVersion` is less than 6.
- **onInsertedSemicolon**: If given a callback, that callback will be called whenever a missing semicolon is inserted by the parser. The callback will be given the character offset of the point where the semicolon is inserted as argument, and if `locations` is on, also a `{line, column}` object representing this position.
- **onTrailingComma**: Like `onInsertedSemicolon`, but for trailing commas.
- **allowReserved**: If `false`, using a reserved word will generate an error. Defaults to `true` for `ecmaVersion` 3, `false` for higher versions. When given the value `"never"`, reserved words and keywords can also not be used as property names (as in Internet Explorer's old parser).
- **allowReturnOutsideFunction**: By default, a return statement at the top level raises an error. Set this to `true` to accept such code.
- **allowImportExportEverywhere**: By default, `import` and `export` declarations can only appear at a program's top level. Setting this option to `true` allows them anywhere where a statement is allowed, and also allows `import.meta` expressions to appear in scripts (when `sourceType` is not `"module"`).
- **allowAwaitOutsideFunction**: If `false`, `await` expressions can only appear inside `async` functions. Defaults to `true` in modules for `ecmaVersion` 2022 and later, `false` for lower

versions. Setting this option to `true` allows to have top-level `await` expressions. They are still not allowed in non-`async` functions, though.

- **allowSuperOutsideMethod**: By default, `super` outside a method raises an error. Set this to `true` to accept such code.
- **allowHashBang**: When this is enabled, if the code starts with the characters `#!` (as in a shellscript), the first line will be treated as a comment. Defaults to true when `ecmaVersion` >= 2023.
- **checkPrivateFields**: By default, the parser will verify that private properties are only used in places where they are valid and have been declared. Set this to false to turn such checks off.
- **locations**: When `true`, each node has a `loc` object attached with `start` and `end` subobjects, each of which contains the one-based line and zero-based column numbers in `{line, column}` form. Default is `false`.
- **onToken**: If a function is passed for this option, each found token will be passed in same format as tokens returned from `tokenizer().getToken()`.

  If array is passed, each found token is pushed to it.

  Note that you are not allowed to call the parser from the callback—that will corrupt its internal state.
- **onComment**: If a function is passed for this option, whenever a comment is encountered the function will be called with the following parameters:
  - `block`: `true` if the comment is a block comment, false if it is a line comment.
  - `text`: The content of the comment.
  - `start`: Character offset of the start of the comment.



based line and zero-based column numbers in `{line, column}` form.

- `--allow-hash-bang`: If the code starts with the characters #! (as in a shellscript), the first line will be treated as a comment.
- `--allow-await-outside-function`: Allows top-level `await` expressions. See the `allowAwaitOutsideFunction` option for more information.
- `--compact`: No whitespace is used in the AST output.
- `--silent`: Do not output the AST, just return the exit status.
- `--help`: Print the usage information and quit.

  The utility spits out the syntax tree as JSON data.

# Existing plugins

- `acorn-jsx`: Parse Facebook JSX syntax extensions

When extending the parser with plugins, you need to call these methods on the extended version of the class. To extend a parser with plugins, you can use its static `extend` method.

```
var acorn = require("acorn");
var jsx = require("acorn-jsx");
var JSXParser =
        acorn.Parser.extend(jsx());
JSXParser.parse("foo(<bar/>)",
        {ecmaVersion: 2020});
```

The `extend` method takes any number of plugin values, and returns a new `Parser` class that includes the extra parser logic provided by the plugins.

## Command line interface

The `bin/acorn` utility can be used to parse a file from the command line. It accepts as arguments its input file and the following options:

- `--ecma3|--ecma5|--ecma6|--ecma7|--ecma8|--ecma9|--ecma10`: Sets the ECMAScript version to parse. Default is version 9.
- `--module`: Sets the parsing mode to `"module"`. Is set to `"script"` otherwise.
- `--locations`: Attaches a “loc” object to each node with “start” and “end” subobjects, each of which contains the one-



  - `end`: Character offset of the end of the comment.

When the `locations` options is on, the `{line, column}` locations of the comment’s start and end are passed as two additional parameters.

If array is passed for this option, each found comment is pushed to it as object in Esprima format:

```
{
  "type": "Line" | "Block",
  "value": "comment text",
  "start": Number,
  "end": Number,
  // If `locations` option is on:
  "loc": {
    "start": {line: Number, column: Number}
    "end": {line: Number, column: Number}
  },
  // If `ranges` option is on:
  "range": [Number, Number]
}
```

Note that you are not allowed to call the parser from the callback—that will corrupt its internal state.

- **ranges**: Nodes have their start and end characters offsets recorded in `start` and `end` properties (directly on the node, rather than the `loc` object, which holds line/column data. To also add a semi-standardized `range` property holding a `[start, end]` array with the same numbers, set the `ranges` option to `true`.

- **program**: It is possible to parse multiple files into a single AST by passing the tree produced by parsing the first file as the `program` option in subsequent parses. This will add the toplevel forms of the parsed file to the "Program" (top) node of an existing parse tree.
- **sourceFile**: When the `locations` option is `true`, you can pass this option to add a `source` attribute in every node's `loc` object. Note that the contents of this option are not examined or processed in any way; you are free to use whatever format you choose.
- **directSourceFile**: Like `sourceFile`, but a `sourceFile` property will be added (regardless of the `location` option) directly to the nodes, rather than the `loc` object.
- **preserveParens**: If this option is `true`, parenthesized expressions are represented by (non-standard) `ParenthesizedExpression` nodes that have a single `expression` property containing the expression inside parentheses.

**parseExpressionAt**`(input, offset, options)` will parse a single expression in a string, and return its AST. It will not complain if there is more of the string left after the expression.

**tokenizer**`(input, options)` returns an object with a `getToken` method that can be called repeatedly to get the next token, a `{start, end, type, value}` object (with added `loc` property when the `locations` option is enabled and `range` property when the `ranges` option is enabled). When the token's type is `tokTypes.eof`, you should stop calling the method, since it will keep returning that same token forever.



Note that tokenizing JavaScript without parsing it is, in modern versions of the language, not really possible due to the way syntax is overloaded in ways that can only be disambiguated by the parse context. This package applies a bunch of heuristics to try and do a reasonable job, but you are advised to use `parse` with the `onToken` option instead of this.

In ES6 environment, returned result can be used as any other protocol-compliant iterable:

```
for (let token of acorn.tokenizer(str))
        {
  // iterate over the tokens
}

// transform code to array of tokens:
var tokens = [...acorn.tokenizer(str)];
```

**tokTypes** holds an object mapping names to the token type objects that end up in the `type` properties of tokens.

**getLineInfo**`(input, offset)` can be used to get a `{line, column}` object for a given program string and offset.

## The `Parser` class

Instances of the `Parser` class contain all the state and logic that drives a parse. It has static methods `parse`, `parseExpressionAt`, and `tokenizer` that match the top-level functions by the same name.