Acorn

A tiny, fast JavaScript parser, written completely 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. For questions and discussion, please use the Tern discussion forum.

Installation

The easiest way to install acorn is with `npm`.

```
npm install acorn
```

Alternately, download the source.

```
git clone https://github.com/ternjs/acorn.git
```

Components

When run in a CommonJS (node.js) or AMD environment, exported values appear in the interfaces exposed by the individual files, as usual. When loaded in the browser (Acorn works in any JS-enabled browser more recent than IE5) without any kind of module management, a single global object `acorn` will be defined, and all the exported properties will be added to that.

Main parser

This is implemented in `dist/acorn.js`, and is what you get when you `require("acorn")` in node.js.

parse `(input, options)` is used to parse a JavaScript program. The `input` parameter is a string, `options` can be undefined or 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.

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 character offset at which the error occurred, and a `loc` object that contains a `{line, column}` object referring to that same position.

- **ecmaVersion**: Indicates the ECMAScript version to parse. Must be either 3, 5, 6 (2015), 7 (2016), or 8 (2017). This influences support for strict mode, the set of reserved words, and support for new syntax features. Default is 7.

  NOTE: Only 'stage 4' (finalized) ECMAScript features are being implemented by Acorn.

- **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.
- **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.
- **allowHashBang**: When this is enabled (off by default), if the code starts with the characters `#!` (as in a shellscript), the first line will be treated as a comment.
- **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.
  - `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.

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

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.

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.

Note on using with [Escodegen]

Escodegen supports generating comments from AST, attached in Esprima-specific format. In order to simulate same format in Acorn, consider following example:

```
var comments = [], tokens = [];

var ast = acorn.parse('var x = 42; // answer', {
    // collect ranges for each node
    ranges: true,
    // collect comments in Esprima's format
    onComment: comments,
    // collect token ranges
    onToken: tokens
});

// attach comments using collected information
escodegen.attachComments(ast, comments, tokens);

// generate code
console.log(escodegen.generate(ast, {comment: true}));
// > 'var x = 42;    // answer'
```

dist/acorn_loose.js

This file implements an error-tolerant parser. It exposes a single function. The loose parser is accessible in node.js via `require("acorn/dist/acorn_loose")`.

parse_dammit `(input, options)` takes the same arguments and returns the same syntax tree as the `parse` function in `acorn.js`, but never raises an error, and will do its best to parse syntactically invalid code in as meaningful a way as it can. It'll insert identifier nodes with name `"✖"` as placeholders in places where it can't make sense of the input. Depends on `acorn.js`, because it uses the same tokenizer.

dist/walk.js

Implements an abstract syntax tree walker. Will store its interface in `acorn.walk` when loaded without a module system.

simple `(node, visitors, base, state)` does a 'simple' walk over a tree. `node` should be the AST node to walk, and `visitors` an object with properties whose names correspond to node types in the ESTree spec. The properties should contain functions that will be called with the node object and, if applicable the state at that point. The last two arguments are optional. `base` is a walker algorithm, and `state` is a start state. The default walker will simply visit all statements and expressions and not produce a meaningful state. (An example of a use of state is to track scope at each point in the tree.)

ancestor `(node, visitors, base, state)` does a 'simple' walk over a tree, building up an array of ancestor nodes (including the current node) and passing the array to the callbacks as a third parameter.

recursive `(node, state, functions, base)` does a 'recursive' walk, where the walker functions are responsible for continuing the walk on the child nodes of their target node. `state` is the start state, and `functions` should contain an object that maps node types to walker functions. Such functions are called with `(node, state, c)` arguments, and can cause the walk to continue on a sub-node by calling the `c` argument

on it with `(node, state)` arguments. The optional `base` argument provides the fallback walker functions for node types that aren't handled in the `functions` object. If not given, the default walkers will be used.

make `(functions, base)` builds a new walker object by using the walker functions in `functions` and filling in the missing ones by taking defaults from `base`.

findNodeAt `(node, start, end, test, base, state)` tries to locate a node in a tree at the given start and/or end offsets, which satisfies the predicate `test`. `start` and `end` can be either `null` (as wildcard) or a number. `test` may be a string (indicating a node type) or a function that takes `(nodeType, node)` arguments and returns a boolean indicating whether this node is interesting. `base` and `state` are optional, and can be used to specify a custom walker. Nodes are tested from inner to outer, so if two nodes match the boundaries, the inner one will be preferred.

findNodeAround `(node, pos, test, base, state)` is a lot like `findNodeAt`, but will match any node that exists 'around' (spanning) the given position.

findNodeAfter `(node, pos, test, base, state)` is similar to `findNodeAround`, but will match all nodes *after* the given position (testing outer nodes before inner nodes).

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` : Sets the ECMAScript version to parse. Default is version 5.
- `--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-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.
- `--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.

Build system

Acorn is written in ECMAScript 6, as a set of small modules, in the project's `src` directory, and compiled down to bigger ECMAScript 3 files in `dist` using Browserify and Babel. If you are already using Babel, you can consider including the modules directly.

The command-line test runner (`npm test`) uses the ES6 modules. The browser-based test page (`test/index.html`) uses the compiled modules. The `bin/build-acorn.js` script builds the latter from the former.

If you are working on Acorn, you'll probably want to try the code out directly, without an intermediate build step. In your scripts, you can register the Babel require shim like this:

```
require("babel-core/register")
```

That will allow you to directly `require` the ES6 modules.

Plugins

Acorn is designed support allow plugins which, within reasonable bounds, redefine the way the parser works. Plugins can add new token types and new tokenizer contexts (if necessary), and extend methods in the parser object. This is not a clean, elegant API—using it requires an understanding of Acorn's internals, and plugins are likely to break whenever those internals are significantly changed. But still, it is *possible*, in this way, to create parsers for JavaScript dialects without forking all of Acorn. And in principle it is even possible to combine such plugins, so that if you have, for example, a plugin for parsing types and a plugin for parsing JSX-style XML literals, you could load them both and parse code with both JSX tags and types.

A plugin should register itself by adding a property to `acorn.plugins`, which holds a function. Calling `acorn.parse`, a `plugins` option can be passed, holding an object mapping plugin names to configuration values (or just `true` for plugins that don't take options). After the parser object has been created, the initialization functions for the chosen plugins are called with `(parser, configValue)` arguments. They are expected to use the `parser.extend` method to extend parser methods. For example, the `readToken` method could be extended like this:

```
parser.extend("readToken", function(nextMethod) {
  return function(code) {
    console.log("Reading a token!")
    return nextMethod.call(this, code)
  }
})
```

The `nextMethod` argument passed to `extend`'s second argument is the previous value of this method, and should usually be called through to whenever the extended method does not handle the call itself.

Similarly, the loose parser allows plugins to register themselves via `acorn.pluginsLoose`. The extension mechanism is the same as for the normal parser:

```
looseParser.extend("readToken", function(nextMethod) {
  return function() {
    console.log("Reading a token in the loose parser!")
    return nextMethod.call(this)
  }
})
```

Existing plugins

- `acorn-jsx` : Parse Facebook JSX syntax extensions
- `acorn-es7-plugin` : Parse async/await syntax proposal
- `acorn-object-spread` : Parse object spread syntax proposal
- `acorn-es7` : Parse decorator syntax proposal

- `acorn-objj` : Objective-J language parser built as Acorn plugin