

Acorn AST walker

An abstract syntax tree walker for the [ESTree](#) format.

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 from [npm](#) :

```
npm install acorn-walk
```

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

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

Interface

An algorithm for recursing through a syntax tree is stored as an object, with a property for each tree node type holding a function that will recurse through such a node. There are several ways to run such a walker.

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.)

```
const acorn = require("acorn")
const walk = require("acorn-walk")

walk.simple(acorn.parse("let x = 10"), {
  Literal(node) {
    console.log(`Found a literal: ${node.value}`)
  }
})
```

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.

```
const acorn = require("acorn")
const walk = require("acorn-walk")
```

```

walk.ancestor(acorn.parse("foo('hi')"), {
  Literal(_, ancestors) {
    console.log("This literal's ancestors are:", ancestors.map(n => n.type))
  }
})

```

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`.

full (node, callback, base, state) does a 'full' walk over a tree, calling the callback with the arguments (node, state, type) for each node

fullAncestor (node, callback, base, state) does a 'full' 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.

```

const acorn = require("acorn")
const walk = require("acorn-walk")

walk.full(acorn.parse("1 + 1"), node => {
  console.log(`There's a ${node.type} node at ${node.ch}`)
})

```

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).