Selectors

Some rules and APIs allow the use of selectors to query an AST. This page is intended to:

1. Explain what selectors are
2. Describe the syntax for creating selectors
3. Describe what selectors can be used for

What is a selector?

A selector is a string that can be used to match nodes in an Abstract Syntax Tree (AST). This is useful for describing a particular syntax pattern in your code.

The syntax for AST selectors is similar to the syntax for CSS selectors. If you've used CSS selectors before, the syntax for AST selectors should be easy to understand.

The simplest selector is just a node type. A node type selector will match all nodes with the given type. For example, consider the following program:

```
var foo = 1;
bar.baz();
```

The selector "`Identifier`" will match all `Identifier` nodes in the program. In this case, the selector will match the nodes for `foo`, `bar`, and `baz`.

Selectors are not limited to matching against single node types. For example, the selector `VariableDeclarator > Identifier` will match all `Identifier` nodes that have a `VariableDeclarator` as a direct parent. In the program above, this will match the node for `foo`, but not the nodes for `bar` and `baz`.

What syntax can selectors have?

The following selectors are supported:

- AST node type: `ForStatement`
- wildcard (matches all nodes): `*`
- attribute existence: `[attr]`
- attribute value: `[attr="foo"]` or `[attr=123]`
- attribute regex: `[attr=/foo.*/]`
- attribute conditions: `[attr!="foo"]`, `[attr>2]`, `[attr<3]`, `[attr>=2]`, or `[attr<=3]`
- nested attribute: `[attr.level2="foo"]`
- field: `FunctionDeclaration > Identifier.id`
- First or last child: `:first-child` or `:last-child`
- nth-child (no ax+b support): `:nth-child(2)`
- nth-last-child (no ax+b support): `:nth-last-child(1)`
- descendant: `FunctionExpression ReturnStatement`
- child: `UnaryExpression > Literal`
- following sibling: `ArrayExpression > Literal + SpreadElement`
- adjacent sibling: `VariableDeclaration ~ VariableDeclaration`
- negation: `:not(ForStatement)`
- matches-any: `:matches([attr] > :first-child, :last-child)`
- class of AST node: `:statement`, `:expression`, `:declaration`, `:function`, or `:pattern`

This syntax is very powerful, and can be used to precisely select many syntactic patterns in your code.

The examples in this section were adapted from the esquery documentation.

What can selectors be used for?

If you're writing custom ESLint rules, you might be interested in using selectors to examine specific parts of the AST. If you're configuring ESLint for your codebase, you might be interested in restricting particular syntax patterns with selectors.

Listening for selectors in rules

When writing a custom ESLint rule, you can listen for nodes that match a particular selector as the AST is traversed.

```
module.exports = {
  create(context) {
    // ...

    return {

      // This listener will be called for all IfStatement nodes with blocks.
      "IfStatement > BlockStatement": function(blockStatementNode) {
        // ...your logic here
      },

      // This listener will be called for all function declarations with more than 3 parameters.
      "FunctionDeclaration[params.length>3]": function(functionDeclarationNode) {
        // ...your logic here
      }
    };
  }
};
```

Adding `:exit` to the end of a selector will cause the listener to be called when the matching nodes are exited during traversal, rather than when they are entered.

If two or more selectors match the same node, their listeners will be called in order of increasing specificity. The specificity of an AST selector is similar to the specificity of a CSS selector:

- When comparing two selectors, the selector that contains more class selectors, attribute selectors, and pseudo-class selectors (excluding `:not()`) has higher specificity.
- If the class/attribute/pseudo-class count is tied, the selector that contains more node type selectors has higher specificity.

If multiple selectors have equal specificity, their listeners will be called in alphabetical order for that node.

Restricting syntax with selectors

With the no-restricted-syntax rule, you can restrict the usage of particular syntax in your code. For example, you can use the following configuration to disallow using `if` statements that do not have block statements as their body:

```
{
  "rules": {
    "no-restricted-syntax": ["error", "IfStatement > :not(BlockStatement).consequent"]
  }
}
```

…or equivalently, you can use this configuration:

```
{
  "rules": {
    "no-restricted-syntax": ["error", "IfStatement[consequent.type!='BlockStatement']"]
  }
}
```

As another example, you can disallow calls to `require()`:

```
{
  "rules": {
    "no-restricted-syntax": ["error", "CallExpression[callee.name='require']"]
  }
}
```

Or you can enforce that calls to `setTimeout` always have two arguments:

```
{
  "rules": {
    "no-restricted-syntax": ["error", "CallExpression[callee.name='setTimeout'][arguments.length!=2]"]
  }
}
```

Using selectors in the `no-restricted-syntax` rule can give you a lot of control over problematic patterns in your codebase, without needing to write custom rules to detect each pattern.
