Esprima

Release 3.1

Jun 06, 2017

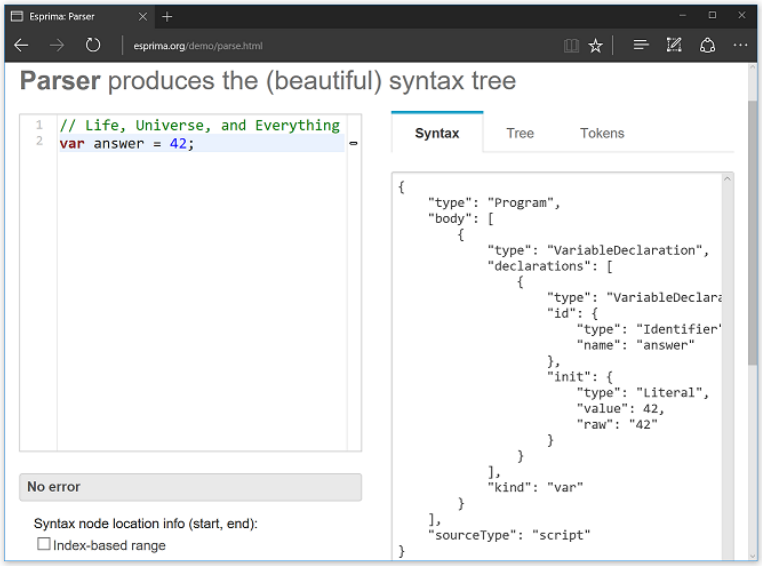
Contents

CHAPTER 1

Chapter 1. Getting Started

Esprima is a tool to perform lexical and syntactical analysis of JavaScript programs. Esprima itself is also written in JavaScript.

To get the feeling of what Esprima can do, please try its online parser demo:



The left panel is a code editor, it can accept any JavaScript source. If the source is a valid JavaScript program, the right panel shows the syntax tree as the result of parsing that JavaScript program. The syntax tree can be displayed in its original form (JSON, but formatted) or in its visual form (hierarchical node view).

Supported environments

Since Esprima is written in JavaScript, it can run on various JavaScript environments, including (but not limited to):

- Modern web browsers (the latest version of Edge, Firefox, Chrome, Safari, etc)
- Legacy web browsers (Internet Explorer 10 or later)
- Node.js v4 or later
- JavaScript engine in Java, such as Rhino and Nashorn

Using Node.js to play with Esprima

To quickly experiment with Esprima, it is recommended to use Node.js and its interactive REPL.

First, install Node.js v4 or later (including its package manager, npm). Then, from the command-line or the terminal, install Esprima npm module:

```
$ npm install esprima
```

To verify that the module is available, use `npm ls`:

```
$ npm ls
/home/ariya/demo
- esprima@3.1.3
```

The number after the `@` symbol, `3.1.3`, indicates the version of Esprima package downloaded and installed from the package registry. This may vary from time to time, depending on the latest stable version available for everyone.

To play with Esprima within Node.js, first launch Node.js. Inside its REPL command prompt, load the module using `require` and then use it, as illustrated in the following session:

```
$ node
> var esprima = require('esprima')
undefined
> esprima.parse('answer = 42')
Program {
  type: 'Program',
  body: [ ExpressionStatement { type: 'ExpressionStatement', expression: [Object] } ],
  sourceType: 'script' }
```

In the above example, the `parse` function of Esprima is invoked with a string containing a simple JavaScript program. The output is an object (printed by Node.js REPL) that is the representation of the program (an assignment expression). This object follows the format described in details in Appendix A. Syntax Tree Format.

If the source given to Esprima parser is not a valid JavaScript program, an exception will be thrown instead. The following example demonstrates that:

```
$ node
> var esprima = require('esprima')
undefined
> esprima.parse('1+')
Error: Line 1: Unexpected end of input
```

To use Esprima in a library or an application designed to be used with Node.js, include `esprima` as a dependency in the `package.json` manifest.

Using Esprima in a web browser

To use Esprima in a browser environment, it needs to be loaded using the `script` element. For instance, to load Esprima from a CDN such as unpkg, the HTML document needs to have the following line:

```
<script src="https://unpkg.com/esprima@~3.1/dist/esprima.js"></script>
```

When Esprima is loaded this way, it will be available as a global object named `esprima`.

Since Esprima supports AMD (Asynchronous Module Definition), it can be loaded with a module loader such as RequireJS:

```
require(['esprima'], function (parser) {
    // Do something with parser, e.g.
    var syntax = parser.parse('var answer = 42');
    console.log(JSON.stringify(syntax, null, 4));
});
```

Using Esprima with Rhino or Nashorn

With Rhino or Nashorn, Esprima must be loaded from its source using the `load` function, such as:

```
load('/path/to/esprima.js');
```

The module `esprima` will be available as part of the global object.

The following session with Nashorn shell, jrunscript, demonstrates the usage:

```
$ jrunscript
nashorn> load('esprima.js')
nashorn> ast = esprima.parse('const answer = 42')
[object Object]
nashorn> print(JSON.stringify(ast, null, 2))
{
  "type": "Program",
  "body": [
    {
      "type": "VariableDeclaration",
      "declarations": [
        {
          "type": "VariableDeclarator",
          "id": {
            "type": "Identifier",
            "name": "answer"
          },
          "init": {
            "type": "Literal",
            "value": 42,
            "raw": "42"
          }
        }
      ],
      "kind": "const"
    }
  ],
```

```
  "sourceType": "script"
}
```

CHAPTER 2

Chapter 2. Syntactic Analysis (Parsing)

The main use case of Esprima is to parse a JavaScript program. This is also known as *syntactic analysis*.

Esprima parser takes a string representing a valid JavaScript program and produces a syntax tree, an ordered tree that describes the syntactic structure of the program. The resulting syntax tree is useful for various purposes, from program transformation to static program analysis.

The interface of the `parse` function is as follows:

```
esprima.parse(input, config, delegate)
```

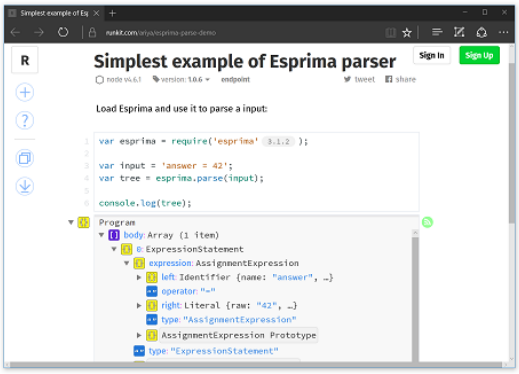
where

- `input` is the string representing the program to be parsed
- `config` is an object used to customize the parsing behavior (optional)
- `delegate` is a callback function invoked for every single node (optional)

The `input` argument is mandatory. Its type must be a string, otherwise the parsing behavior is not determined. The other two arguments, `config` and `delegate`, are optional.

The object returned by the `parse` function is the syntax tree, following the format described in details in Appendix A. Syntax Tree Format.

The description of various parsing configuration is summarized in the following table:



The previous chapter, Getting Started, already demonstrates the simplest possible example of using Esprima parser. To use it as the basis for further experiments, use Runkit and tweak the existing demo notebook: runkit.com/ariya/esprima-parse-demo.

Distinguishing a Script and a Module

With ES2015 and later, a JavaScript program can be either a script or a module. It is a very important distinction, a parser such as Esprima needs to know the type of the source to be able to analyze its syntax correctly. This is achieved by specifying the `sourceType` property in the configuration object. The default value is `"script"`, i.e. the program will be treated as a script and not a module. Another possible value is `"module"`, which instructs the parser to treat the program as a module.

An example of parsing a script:

```
$ node
> var esprima = require('esprima')
> esprima.parse('answer = 42', { sourceType: 'script' });
Program {
  type: 'Program',
  body: [ ExpressionStatement { type: 'ExpressionStatement', expression: [Object] } ],
  sourceType: 'script' }
```

An example of parsing a module:

```
$ node
> var esprima = require('esprima')
> esprima.parse('import { sqrt } from "math.js"', { sourceType: 'module' });
Program {
  type: 'Program',
  body:
   [ ImportDeclaration {
       type: 'ImportDeclaration',
       specifiers: [Object],
       source: [Object] } ],
  sourceType: 'module' }
```

Failing to specify the source type can lead to a *mistaken* observation that Esprima does not support a certain syntax. Take a look at this example:

```
$ node
> var esprima = require('esprima')
> esprima.parse('export const answer = 42');
Error: Line 1: Unexpected token
```

Instead of producing the syntax tree, the parser throws an exception. This is the correct behavior, an `export` statement can only be part of a module, not a script. Thus, the parser properly determines that such a source program (treated as a script since `sourceType` is not specified) is invalid.

JSX Syntax Support

JSX is a syntax extension to JavaScript, popularly known to build web applications using React. JSX is not part of any official ECMAScript specification. Application code using JSX is typically preprocessed first into standard JavaScript.

Esprima parser fully understands JSX syntax when `jsx` flag in the parsing configuration object is set to true, as illustrated in the following example:

```
$ node
> var esprima = require('esprima')
> esprima.parse('var el= <title>${product}</title>', { jsx: true });
Program {
  type: 'Program',
  body:
   [ VariableDeclaration {
       type: 'VariableDeclaration',
       declarations: [Object],
       kind: 'var' } ],
  sourceType: 'script' }
```

Browser Compatibility

In a certain specific case, Esprima parser *intentionally* does not throw an exception (indicating a syntax error) although the input being parsed is not valid. This is to achieve an implementation compatibility with major web browsers. For further details, refer to the official ECMAScript 2015 Language Specification, Section B.3.3 on **Block-Level Function Declarations Web Legacy Compatibility Semantics**:

> Prior to ECMAScript 2015, the ECMAScript specification did not define the occurrence of a *FunctionDeclaration* as an element of a *Block* statement's *StatementList*. However, support for that form of *FunctionDeclaration* was an allowable extension and most browser-hosted ECMAScript implementations permitted them.

This is illustrated in the following example:

```
$ node
> var esprima = require('esprima')
> esprima.parse('if (x) function y() {}')
Program {
  type: 'Program',
  body:
   [ IfStatement {
       type: 'IfStatement',
       test: [Object],
       consequent: [Object],
       alternate: null } ],
  sourceType: 'script' }
```

In the above example, Esprima parser returns a syntax tree for the code (it does not throw an exception) even though the input is invalid according to the specification, i.e. declaring a function inside the block of an If statement is not possible. In this case, the behavior of Esprima is the same as the popular web browsers such as Firefox, Chrome, Safari, and many others.

Tolerant Mode

When Esprima parser is given an input that does not represent a valid JavaScript program, it throws an exception. With the tolerant mode however, the parser *may* choose to continue parsing and produce a syntax tree. This is best demonstrated with an example.

Consider the following parsing session. The exception is expected, since a `with` statement is not permitted in strict mode.

```
$ node
> var esprima = require('esprima')
> esprima.parse('"use strict"; with (x) {}')
Error: Line 1: Strict mode code may not include a with statement
```

If the tolerant mode is activated by setting the `tolerant` flag to true in the parsing configuration, the parser behaves differently:

```
$ node
> var esprima = require('esprima')
> esprima.parse('"use strict"; with (x) {}', { tolerant: true })
Program {
  type: 'Program',
  body:
   [ Directive {
       type: 'ExpressionStatement',
       expression: [Object],
       directive: 'use strict' },
     WithStatement { type: 'WithStatement', object: [Object], body: [Object] } ],
  sourceType: 'script',
  errors:
   [ { Error: Line 1: Strict mode code may not include a with statement
       index: 13,
       lineNumber: 1,
       description: 'Strict mode code may not include a with statement' } ] }
```

In the above case, the parser does not throw an exception and it still returns a syntax tree. However, it also adds a new array named `errors`, it contains each and every syntax error that the parser manages to tolerate without causing it to stop right away. Each entry in the `errors` has the detailed information regarding the error, including the description and the location.

Note that the tolerant mode is intended to deal with very few types of syntax errors. It is unable to robustly handle every possible invalid program.

Node Location

By default, Esprima parser produces an abstract syntax tree. For some uses cases, this abstract syntax tree is not sufficient. For instance, having the location information of each node is necessary in a few cases of static analysis, e.g. to give a meaningful feedback to the user.

To have each node carries some additional properties indicating its location, the parser must be invoked by specifying the flags, `range` or `loc` or both of them, in the parsing configuration.

Setting the `range` flag to true adds a new property, `range`, to each node. It is an array of two elements, each indicating the zero-based index of the starting and end location (exclusive) of the node. A simple example follows:

```
$ node
> var esprima = require('esprima')
> esprima.parse('answer = 42', { range: true })
Program {
  type: 'Program',
  body:
   [ ExpressionStatement {
       type: 'ExpressionStatement',
       expression: [Object],
       range: [Object] } ],
  sourceType: 'script',
  range: [ 0, 11 ] }
```

The top-level *Program* node has the range [0, 11]. This indicates that the program starts from the zeroth character and finishes before the 11th character.

A subsequent example to inspect the numeric literal 42 on the right side of the assignment:

```
$ node
> var esprima = require('esprima')
> var program = esprima.parse('answer = 42', { range: true })
> program.body[0].expression.right
Literal { type: 'Literal', value: 42, raw: '42', range: [ 9, 11 ] }
```

In the above example, the location of the *Literal* node is determined to be [9, 11]. In other words, the ninth and the tenth characters (the eleventh is excluded) of the source correspond to the numeric literal `42`.

Setting the `loc` flag to true adds a new property, `loc`, to each node. It is a object that contains the line number and column number of the starting and end location (exclusive) of the node. It is illustrated in this example:

```
$ node
> var esprima = require('esprima')
> esprima.parse('answer = 42', { loc: true })
Program {
  type: 'Program',
  body:
   [ ExpressionStatement {
       type: 'ExpressionStatement',
       expression: [Object],
       loc: [Object] } ],
  sourceType: 'script',
  loc: { start: { line: 1, column: 0 }, end: { line: 1, column: 11 } } }
```

Note that the line number is *one-based* while the column number is *zero-based*.

It is possible to set both `range` and `loc` to true, thereby giving each token the most complete location information.

Handling Hashbang/Shebang

In a Unix environment, a shell script often has its first line marked by a hashbang or a shebang, `#!`. A common example is a utility intended to be executed by Node.js, which may look like the following:

```
#!/usr/bin/env node
console.log('Hello from Node.js!');
```

If Esprima parser is being used to process the content of the above file, the parser will throw an exception. This is because that hashbang is not valid in JavaScript. A quick Node.js REPL session to illustrate the point:

```
$ node
> var esprima = require('esprima')
> var src = ['#!/usr/bin/env node', 'answer = 42'].join('\n')
> esprima.parse(src)
Error: Line 1: Unexpected token ILLEGAL
```

The workaround for this problem is to remove the first line completely before passing it to the parser. One way to do that is to use a regular expression, as shown below:

```
$ node
> var esprima = require('esprima')
> var src = ['#!/usr/bin/env node', 'answer = 42'].join('\n')
> src = src.replace(/^#!(.*\n)/, '')
'answer = 42'
> esprima.parse(src)
Program {
  type: 'Program',
  body: [ ExpressionStatement { type: 'ExpressionStatement', expression: [Object] } ],
  sourceType: 'script' }
```

Note that the above approach will shorten the source string. If the string length needs to be preserved, e.g. to facilitate an exact location mapping to the original version, then a series of whitespaces need to be padded to the beginning. A modified approach looks like the following:

```
$ node
> var esprima = require('esprima')
> var src = ['#!/usr/bin/env node', 'answer = 42'].join('\n')
> src = src.replace(/(^#!.*)/, function(m) { return Array(m.length + 1).join(' ') });
> esprima.parse(src, { range: true })
Program {
  type: 'Program',
  body:
   [ ExpressionStatement {
       type: 'ExpressionStatement',
       expression: [Object],
       range: [Object] } ],
  sourceType: 'script',
  range: [ 15, 26 ] }
```

Token Collection

When Esprima parser is performing the syntactical analysis, first it needs to break down the source into a series of tokens. By default, the tokens are not stored as part of the parsing result. It is possible to keep the tokens found during the parsing by setting the `tokens` flag in the configuration object to true. Take a look at this example:

```
$ node
> var esprima = require('esprima')
> esprima.parse('const answer = 42', { tokens: true })
Program {
```

```
  type: 'Program',
  body:
   [ VariableDeclaration {
       type: 'VariableDeclaration',
       declarations: [Object],
       kind: 'const' } ],
  sourceType: 'script',
  tokens:
   [ { type: 'Keyword', value: 'const' },
     { type: 'Identifier', value: 'answer' },
     { type: 'Punctuator', value: '=' },
     { type: 'Numeric', value: '42' } ] }
```

The output of the parser now contains an additional property, an array named `tokens`. Every element in this array is the token found during the parsing process. For each token, the `type` property is a string indicating the type of the token and the `value` property stores the corresponding the *lexeme*, i.e. a string of characters which forms a syntactic unit.

The token also contains its location, if the parsing configuration has the flag `range` or `loc` (or both), as shown in the following example:

```
$ node
> var esprima = require('esprima')
> var output = esprima.parse('const answer = 42', { tokens: true, range: true })
> output.tokens
[ { type: 'Keyword', value: 'const', range: [ 0, 5 ] },
  { type: 'Identifier', value: 'answer', range: [ 6, 12 ] },
  { type: 'Punctuator', value: '=', range: [ 13, 14 ] },
  { type: 'Numeric', value: '42', range: [ 15, 17 ] } ]
```

To tokenize a program without parsing it at all, refer to Chapter 3. Lexical Analysis (Tokenization).

Comment Collection

Comments do not affect the syntax of a JavaScript program and therefore they are ignored by Esprima parser. However, if it is important to collect every single-line and multi-line comment in the program, Esprima parser can be instructed to collect them by setting the `comment` flag in the parsing configuration to true.

Consider the following example. The output of the parser has an additional property, an array named `comments`. Every element in this array indicates a single-line or a mult-line comment encountered during the parsing process.

```
$ node
> var esprima = require('esprima')
> esprima.parse('answer = 42 // TODO: why', { comment: true })
Program {
  type: 'Program',
  body: [ ExpressionStatement { type: 'ExpressionStatement', expression: [Object] } ],
  sourceType: 'script',
  comments: [ { type: 'Line', value: ' TODO: why' } ] }
```

The type of each comment can either be *Line* for a single-line comment (`//` towards the end-of-line) or *Block* for a multi-line comment (enclosed by `/*` and `*/`).

```
$ node
> var esprima = require('esprima')
> esprima.parse('/*everything*/ answer = 42', { comment: true })
```

```
Program {
  type: 'Program',
  body: [ ExpressionStatement { type: 'ExpressionStatement', expression: [Object] } ],
  sourceType: 'script',
  comments: [ { type: 'Block', value: 'everything' } ] }
```

Each comment can also contain its location, if the parsing configuration has the flag `range` or `loc` (or both), illustrated here:

```
$ node
> var esprima = require('esprima')
> output = esprima.parse('answer = 42 // TODO: why', { comment: true, range: true });
> output.comments
[ { type: 'Line', value: ' TODO: why', range: [ 12, 24 ] } ]
```

Syntax Delegate

The last argument in the `parse` function is a delegate, a callback function invoked for each syntax node (as the node is constructed) with two arguments, the node object itself and the node metadata. The metadata contains the start and end location of the node.

The shortest Node.js script that illustrates the delegate is the following:

```
var esprima = require('esprima');
esprima.parse('answer = 42', {}, function (node) {
    console.log(node.type);
});
```

If the script is called `delegate.js`, running it with Node.js gives the output:

```
$ node delegate.js
Identifier
Literal
AssignmentExpression
ExpressionStatement
Program
```

In the above example, the callback function only does one simple thing: printing the type of the node. The first two calls are with the deepest nodes (which are also leaves), an *Identifier* node representing `answer` and a *Literal* node for `42`. The next is with an *AssignmentExpression* node that combines the previous two nodes (`answer = 42`). It is then followed by the only statement in the code fragment, an *ExpressionStatement* node. The last one, as with every syntax tree, is always the top-level *Program* node.

The order of the nodes being passed as the argument for the callback function resembles a tree traversal using the depth-first algorithm.

If each single-line and multi-line comment must be passed to the callback function as well, the `comment` flag in the parsing configuration needs to be true. A modified `delegate.js` and its new result (note the addition of *LineComment* node type):

```
$ cat delegate.js
var esprima = require('esprima');
esprima.parse('answer = 42 // FIXME', { comment: true }, function (node) {
    console.log(node.type);
});
```

```
$ node delegate.js
Identifier
LineComment
Literal
AssignmentExpression
ExpressionStatement
Program
```

The second argument passed to the callback function, the metadata, can be used to locate the node. It is illustrated in the following example:

```
$ cat delegate.js
var esprima = require('esprima');
esprima.parse('answer = 42', {}, function (node, metadata) {
    console.log(node.type, metadata);
});

$ node delegate.js
Identifier { start: { line: 1, column: 0, offset: 0 },
  end: { line: 1, column: 6, offset: 6 } }
Literal { start: { line: 1, column: 9, offset: 9 },
  end: { line: 1, column: 11, offset: 11 } }
AssignmentExpression { start: { line: 1, column: 0, offset: 0 },
  end: { line: 1, column: 11, offset: 11 } }
ExpressionStatement { start: { line: 1, column: 0, offset: 0 },
  end: { line: 1, column: 11, offset: 11 } }
Program { start: { line: 1, column: 0, offset: 0 },
  end: { line: 1, column: 11, offset: 11 } }
```

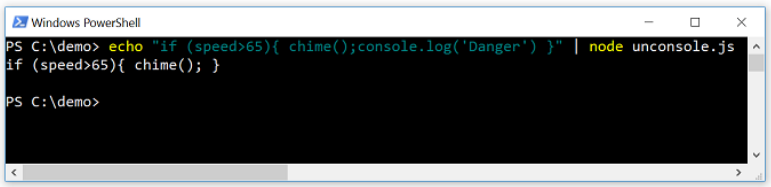
Example: console Calls Removal

The following Node.js script demonstrates the use of Esprima parser to remove every single expression that represents a `console` call. The script accepts the input from `stdin` and displays the result to `stdout`. The entire script comprises approximately 30 lines of code.

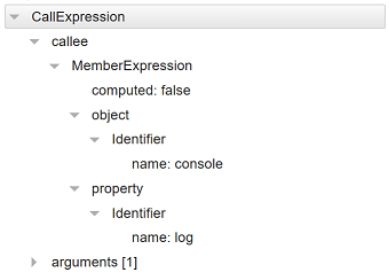
```
const esprima = require('esprima');
const readline = require('readline');

// console.log(x) or console['error'](y)
function isConsoleCall(node) {
    return (node.type === 'CallExpression') &&
        (node.callee.type === 'MemberExpression') &&
        (node.callee.object.type === 'Identifier') &&
        (node.callee.object.name === 'console');
}

function removeCalls(source) {
    const entries = [];
    esprima.parse(source, {}, function (node, meta) {
        if (isConsoleCall(node)) {
            entries.push({
                start: meta.start.offset,
                end: meta.end.offset
            });
```

```
        }
    });
    entries.sort((a, b) => { return b.end - a.end }).forEach(n => {
        source = source.slice(0, n.start) + source.slice(n.end);
    });
    return source;
}

let source = '';
readline.createInterface({ input: process.stdin, terminal: false })
.on('line', line => { source += line + '\n' })
.on('close', () => { console.log(removeCalls(source)) });
```

An example run is shown in the following screenshot (the script is called `unconsole.js`). Note that the single call to `console.log` is eliminated in the output.



The script uses the readline module to read the input line-by-line, collecting each line to a buffer. Once there is no more input, it uses Esprima parser and utilizes the syntax delegate feature with a callback function that looks for a particular type of syntax node, i.e. a call expression with `console` object as the callee. The logic inside the `isConsoleCall` function is intended to match such a node. As an illustration, using the Esprima online demo to parse `console.log("Hello")` will reveal the following syntax tree:



For each matched node, the node location is recorded. Once the parsing is completed, the list of the location of every matched call expression with `console` is used to modify the source, i.e. the portion of the source corresponding to the call expression is removed. When it is done repeatedly, the result is that every single `console` call will disappear. Note how this is done from the last one to the first one (reflected by the sorting in the reverse order) to maintain the correct offset throughout the process.

CHAPTER 3

Chapter 3. Lexical Analysis (Tokenization)

Esprima tokenizer takes a string as an input and produces an array of tokens, a list of object representing categorized input characters. This is known as *lexical analysis*.

The interface of the `tokenize` function is as follows:

```
esprima.tokenize(input, config)
```

where

- `input` is a string representing the program to be tokenized
- `config` is an object used to customize the parsing behavior (optional)

The `input` argument is mandatory. Its type must be a string, otherwise the tokenization behavior is not determined.

The description of various properties of `config` is summarized in the following table:

An example Node.js REPL session that demonstrates the use of Esprima tokenizer is:

```
$ node
> var esprima = require('esprima')
> esprima.tokenize('answer = 42')
[ { type: 'Identifier', value: 'answer' },
  { type: 'Punctuator', value: '=' },
  { type: 'Numeric', value: '42' } ]
```

In the above example, the input string is tokenized into 3 tokens: an identifier, a punctuator, and a number. For each token, the `type` property is a string indicating the type of the token and the `value` property stores the corresponding the *lexeme*, i.e. a string of characters which forms a syntactic unit.

Unlike the `parse` function, the `tokenize` function can work with an input string that does not represent a valid JavaScript program. This is because lexical analysis, as the name implies, does *not* involve the process of understanding the syntactic structure of the input.

```
$ node
> var esprima = require('esprima')
> esprima.tokenize('42 = answer')
```

```
[ { type: 'Numeric', value: '42' },
  { type: 'Punctuator', value: '=' },
  { type: 'Identifier', value: 'answer' } ]
> esprima.tokenize('while (if {}')
[ { type: 'Keyword', value: 'while' },
  { type: 'Punctuator', value: '(' },
  { type: 'Keyword', value: 'if' },
  { type: 'Punctuator', value: '{' },
  { type: 'Punctuator', value: '}' } ]
```

Token Location

By default, each token in the array returned by the tokenizer only has two properties, the *type* of the token and the *lexeme*. For some use cases, the location of each token needs to be known as well (e.g. to offer a meaningful feedback to the user). Esprima tokenizer can add that location information to each token in two forms, zero-based range and line-column location. This is done by customizing the tokenization process with the configuration object.

Setting `range` (in the configuration object) to true adds a new property, `range`, to each token. It is an array of two elements, each indicating the zero-based index of the *starting* and *end* location (exclusive) of the token. A simple example follows:

```
$ node
> var esprima = require('esprima')
> esprima.tokenize('answer = 42', { range: true })
[ { type: 'Identifier', value: 'answer', range: [ 0, 6 ] },
  { type: 'Punctuator', value: '=', range: [ 7, 8 ] },
  { type: 'Numeric', value: '42', range: [ 9, 11 ] } ]
```

In the above example, the starting and end location of each token can be determined from its range property. For instance, the equal sign (=) is the 7th character in the input string, because its range is `[7, 8]`.

Setting `loc` to true adds a new property, `loc`, to each token. It is a object that contains the *line number* and *column number* of the starting and end location (exclusive) of the token. This is illustrated in the example:

```
$ node
> var esprima = require('esprima')
> tokens = esprima.tokenize('answer = 42', { loc: true });
> tokens[2]
{ type: 'Numeric',
  value: '42',
  loc: { start: { line: 1, column: 9 }, end: { line: 1, column: 11 } } }
```

Note that the line number is *one-based* while the column number is *zero-based*.

It is possible to set both `range` and `loc` to true, thereby giving each token the most complete location information.

Line and Block Comments

By default, Esprima tokenizer ignores every line and block comment. If each comment needs to be included in the output, then the property `comment` in the configuration object needs to be set to true. To illustrate this, compare the following simple tokenization:

```
$ node
> var esprima = require('esprima')
> esprima.tokenize('/* answer */ 42')
[ { type: 'Numeric', value: '42' } ]
```

with the following situation where the token array also contains the block comment:

```
$ node
> var esprima = require('esprima')
> esprima.tokenize('/* answer */ 42', { comment: true })
[ { type: 'BlockComment', value: ' answer ' },
  { type: 'Numeric', value: '42' } ]
```

If the location of each comment is needed, enable the location information using `range` and/or `loc` (as explained in the previous section):

```
$ node
> var esprima = require('esprima')
> esprima.tokenize('/* answer */ 42', { comment: true, range: true })
[ { type: 'BlockComment', value: ' answer ', range: [ 0, 12 ] },
  { type: 'Numeric', value: '42', range: [ 13, 15 ] } ]
```

Limitation on Keywords

Since a tokenization process does not have the context of the syntactic structure, it is unable to infer properly that a particular reserved word is being used not as a keyword. Therefore, it always classifies a reserved word as keyword. A simple example to illustrate this limitation:

```
$ node
> var esprima = require('esprima')
> esprima.tokenize('x.if = 1')
[ { type: 'Identifier', value: 'x' },
  { type: 'Punctuator', value: '.' },
  { type: 'Keyword', value: 'if' },
  { type: 'Punctuator', value: '=' },
  { type: 'Numeric', value: '1' } ]
```

In the above session, the type of the `if` token is `Keyword`.

This is however different than what will be obtained using Esprima parser since the parser correctly matches the `if` token as an object property and therefore constructs an associated `Identifier` node, not a `Keyword` node.

```
$ node
> var esprima = require('esprima')
> esprima.parse('x.if = 1').body[0].expression.left.property
Identifier { type: 'Identifier', name: 'if' }
```

Limitation on JSX

JSX is a syntax extension to JavaScript, popularly known to build web applications using React. JSX is not part of any official ECMAScript specification.

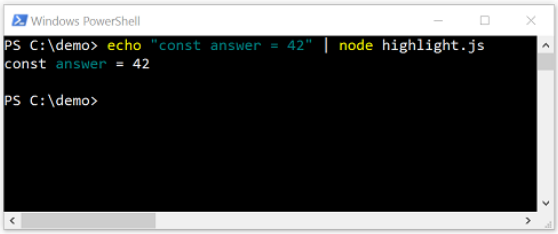
Esprima tokenizer is unable to process input source that contains a mix of JavaScript code and JSX. This is because switching to JSX mode requires an understanding of the context, which a tokenizer does not have. In particular, a closing JSX element (such as `</title>`) confuses the tokenizer since the forward slash (`/`) is identified as the start of a regular expression.

Example: Syntax Highlighting

The following Node.js script demonstrates the use of Esprima tokenizer to apply syntax highlighting of JavaScript code fragment. It accepts the input from `stdin` and produces color coded version to `stdout` by using ANSI escape code.

```
const esprima = require('esprima');
const readline = require('readline');

const CYAN = '\x1b[36m';
const RESET = '\x1b[0m'
let source = '';

readline.createInterface({ input: process.stdin, terminal: false })
.on('line', line => { source += line + '\n' })
.on('close', () => {
    const tokens = esprima.tokenize(source, { range: true });
    const ids = tokens.filter(x => x.type === 'Identifier');
    const markers = ids.sort((a, b) => { return b.range[0] - a.range[0] });
    markers.forEach(t => {
        const id = CYAN + t.value + RESET;
        const start = t.range[0];
        const end = t.range[1];
        source = source.slice(0, start) + id + source.slice(end);
    });
    console.log(source);
});
```

An example run is shown in the following screenshot (the script is called `highlight.js`):



The script uses the readline module to read the input line-by-line, collecting each line to a local string buffer. Once there is no more input, it invokes Esprima tokenizer to break the source into a list of tokens. The script only cares about identifier tokens, hence the filtering. For each token, the starting location is used to determine where to insert the escape code to change the color to cyan and the end location is used to reset the color. This is done from the last identifier token to the first identifier token, which necessities the sorting in reverse order.

For a real-world syntax highlighter that has many more features, take a look at *cardinal* (source repository: github.com/thlorenz/cardinal). It uses a similar approach, i.e. using Esprima tokenizer to break the source into tokens and then wrap each token with a type-specific color.

CHAPTER 4

Appendix A. Syntax Tree Format

Esprima syntax tree format is derived from the original version of Mozilla Parser API, which is then formalized and expanded as the ESTree specification.

Note: In the following sections, interfaces are described using the syntax of TypeScript interface.

Each node is represented as a regular JavaScript object that implements the interface:

```
interface Node {
  type: string;
}
```

The `type` property is a string that contains the variant type of the node. Each subtype of *Node* is explained in the subsequent sections.

When the node is annotated with its location, the interface becomes:

```
interface Node {
  type: string;
  range?: [number, number];
  loc?: SourceLocation;
}
```

with the source location defined as:

```
interface Position {
    line: number;
    column: number;
}

interface SourceLocation {
    start: Position;
    end: Position;
    source?: string | null;
}
```

Expressions and Patterns

A binding pattern can be one of the following:

```
type BindingPattern = ArrayPattern | ObjectPattern;
```

An expression can be one of the following:

```
type Expression = ThisExpression | Identifier | Literal |
    ArrayExpression | ObjectExpression | FunctionExpression | ArrowFunctionExpression
↪| ClassExpression |
    TaggedTemplateExpression | MemberExpression | Super | MetaProperty |
    NewExpression | CallExpression | UpdateExpression | UnaryExpression |
    BinaryExpression | LogicalExpression | ConditionalExpression |
    YieldExpression | AssignmentExpression | SequenceExpression;
```

Array Pattern

```
interface ArrayPattern {
    type: 'ArrayPattern';
    elements: ArrayPatternElement[];
}
```

with

```
type ArrayPatternElement = AssignmentPattern | Identifier | BindingPattern |
↪RestElement | null;

interface RestElement {
    type: 'RestElement';
    argument: Identifier | BindingPattern;
}
```

Assignment Pattern

```
interface AssignmentPattern {
    type: 'AssignmentPattern';
    left: Identifier | BindingPattern;
    right: Expression;
}
```

Object Pattern

```
interface ObjectPattern {
    type: 'ObjectPattern';
    properties: Property[];
}
```

This Expression

```
interface ThisExpression {
    type: 'ThisExpression';
}
```

Identifier

```
interface Identifier {
    type: 'Identifier';
    name: string;
}
```

Literal

```
interface Literal {
    type: 'Literal';
    value: boolean | number | string | RegExp | null;
    raw: string;
    regex?: { pattern: string, flags: string };
}
```

The `regex` property only applies to regular expression literals.

Array Expression

```
interface ArrayExpression {
    type: 'ArrayExpression';
    elements: ArrayExpressionElement[];
}
```

where

```
type ArrayExpressionElement = Expression | SpreadElement;
```

Object Expression

```
interface ObjectExpression {
    type: 'ObjectExpression';
    properties: Property[];
}
```

where

```
interface Property {
    type: 'Property';
    key: Identifier | Literal;
    computed: boolean;
    value: AssignmentPattern | Identifier | BindingPattern | FunctionExpression |␣
↪null;
```

```
    kind: 'get' | 'set' | 'init';
    method: false;
    shorthand: boolean;
}
```

Function Expression

```
interface FunctionExpression {
    type: 'FunctionExpression';
    id: Identifier | null;
    params: FunctionParameter[];
    body: BlockStatement;
    generator: boolean;
    expression: boolean;
}
```

with

```
type FunctionParameter = AssignmentPattern | Identifier | BindingPattern;
```

The value of `generator` is true for a generator expression.

Arrow Function Expression

```
interface FunctionExpression {
    type: 'ArrowFunctionExpression';
    id: Identifier | null;
    params: FunctionParameter[];
    body: BlockStatement | Expression;
    generator: boolean;
    expression: false;
}
```

Class Expression

```
interface ClassExpression {
    type: 'ClassExpression';
    id: Identifier | null;
    superClass: Identifier | null;
    body: ClassBody;
```

with

```
interface ClassBody {
    type: 'ClassBody';
    body: MethodDefinition[];
}

interface MethodDefinition {
    type: 'MethodDefinition';
    key: Expression | null;
    computed: boolean;
```

```
    value: FunctionExpression | null;
    kind: 'method' | 'constructor';
    static: boolean;
}
```

Tagged Template Expression

```
interface TaggedTemplateExpression {
    type: 'TaggedTemplateExpression';
    readonly tag: Expression;
    readonly quasi: TemplateLiteral;
}
```

with

```
interface TemplateElement {
    type: 'TemplateElement';
    value: { cooked: string; raw: string };
    tail: boolean;
}

interface TemplateLiteral {
    type: 'TemplateLiteral';
    quasis: TemplateElement[];
    expressions: Expression[];
}
```

Member Expression

```
interface MemberExpression {
    type: 'MemberExpression';
    computed: boolean;
    object: Expression;
    property: Expression;
}
```

Super

```
interface Super {
    type: 'Super';
}
```

MetaProperty

```
interface MetaProperty {
    type: 'MetaProperty';
    meta: Identifier;
    property: Identifier;
}
```

Call and New Expressions

```
interface CallExpression {
    type: 'CallExpression';
    callee: Expression;
    arguments: ArgumentListElement[];
}

interface NewExpression {
    type: 'NewExpression';
    callee: Expression;
    arguments: ArgumentListElement[];
}
```

with

```
type ArgumentListElement = Expression | SpreadElement;

interface SpreadElement {
    type: 'SpreadElement';
    argument: Expression;
}
```

Update Expression

```
interface UpdateExpression {
    type: 'UpdateExpression';
    operator: '++' | '--';
    argument: Expression;
    prefix: boolean;
}
```

Unary Expression

```
interface UnaryExpression {
    type: 'UnaryExpression';
    operator: '+' | '-' | '~' | '!' | 'delete' | 'void' | 'typeof';
    argument: Expression;
    prefix: true;
}
```

Binary Expression

```
interface BinaryExpression {
    type: 'BinaryExpression';
    operator: 'instanceof' | 'in' | '+' | '-' | '*' | '/' | '%' | '**' |
        '|' | '^' | '&' | '==' | '!=' | '===' | '!==' |
        '<' | '>' | '<=' | '<<' | '>>' | '>>>';
    left: Expression;
    right: Expression;
}
```

Logical Expression

```
interface LogicalExpression {
    type: 'LogicalExpression';
    operator: '||' | '&&';
    left: Expression;
    right: Expression;
}
```

Conditional Expression

```
interface ConditionalExpression {
    type: 'ConditionalExpression';
    test: Expression;
    consequent: Statement;
    alternate?: Statement;
}
```

Yield Expression

```
interface YieldExpression {
    type: 'YieldExpression';
    argument: Expression | null;
    delegate: boolean;
}
```

Assignment Expression

```
interface AssignmentExpression {
    type: 'AssignmentExpression';
    operator: '=' | '*=' | '**=' | '/=' | '%=' | '+=' | '-=' |
        '<<=' | '>>=' | '>>>=' | '&=' | '^=' | '|=';
    left: Expression;
    right: Expression;
}
```

Sequence Expression

```
interface SequenceExpression {
    type: 'SequenceExpression';
    expressions: Expression[];
}
```

Statements and Declarations

A statement can be one of the following:

```
type Statement = BlockStatement | BreakStatement | ContinueStatement |
    DebuggerStatement | DoWhileStatement | EmptyStatement |
    ExpressionStatement | ForStatement | ForInStatement |
    ForOfStatement | FunctionDeclaration | IfStatement |
    LabeledStatement | ReturnStatement | SwitchStatement |
    ThrowStatement | TryStatement | VariableDeclaration |
    WhileStatement | WithStatement;
```

A declaration can be one of the following:

```
type Declaration = ClassDeclaration | FunctionDeclaration  VariableDeclaration;
```

A statement list item is either a statement or a declaration:

```
type StatementListItem = Declaration | Statement;
```

Block Statement

A series of statements enclosed by a pair of curly braces form a block statement:

```
interface BlockStatement {
    type: 'BlockStatement';
    body: StatementListItem[];
}
```

Break Statement

```
interface BreakStatement {
    type: 'BreakStatement';
    label: Identifier | null;
}
```

Class Declaration

```
interface ClassDeclaration {
    type: 'ClassDeclaration';
    id: Identifier | null;
    superClass: Identifier | null;
    body: ClassBody;
}
```

Continue Statement

```
interface ContinueStatement {
    type: 'ContinueStatement';
    label: Identifier | null;
}
```

Debugger Statement

```
interface DebuggerStatement {
    type: 'DebuggerStatement';
}
```

Do-While Statement

```
interface DoWhileStatement {
    type: 'DoWhileStatement';
    body: Statement;
    test: Expression;
}
```

Empty Statement

```
interface EmptyStatement {
    type: 'EmptyStatement';
}
```

Expression Statement

```
interface ExpressionStatement {
    type: 'ExpressionStatement';
    expression: Expression;
    directive?: string;
}
```

When the expression statement represents a directive (such as `"use strict"`), then the `directive` property will contain the directive string.

For Statement

```
interface ForStatement {
    type: 'ForStatement';
    init: Expression | VariableDeclaration | null;
    test: Expression | null;
    update: Expression | null;
    body: Statement;
}
```

For-In Statement

```
interface ForInStatement {
    type: 'ForInStatement';
    left: Expression;
    right: Expression;
    body: Statement;
```

```
    each: false;
}
```

For-Of Statement

```
interface ForOfStatement {
    type: 'ForOfStatement';
    left: Expression;
    right: Expression;
    body: Statement;
}
```

Function Declaration

```
interface FunctionDeclaration {
    type: 'FunctionDeclaration';
    id: Identifier | null;
    params: FunctionParameter[];
    body: BlockStatement;
    generator: boolean;
    expression: false;
}
```

with

```
type FunctionParameter = AssignmentPattern | Identifier | BindingPattern;
```

If Statement

```
interface IfStatement {
    type: 'IfStatement';
    test: Expression;
    consequent: Statement;
    alternate?: Statement;
}
```

Labelled Statement

A statement prefixed by a label becomes a labelled statement:

```
interface LabeledStatement {
    type: 'LabeledStatement';
    label: Identifier;
    body: Statement;
}
```

Return Statement

```
interface ReturnStatement {
    type: 'ReturnStatement';
    argument: Expression | null;
}
```

Switch Statement

```
interface SwitchStatement {
    type: 'SwitchStatement';
    discriminant: Expression;
    cases: SwitchCase[];
}
```

with

```
interface SwitchCase {
    type: 'SwitchCase';
    test: Expression;
    consequent: Statement[];
}
```

Throw Statement

```
interface ThrowStatement {
    type: 'ThrowStatement';
    argument: Expression;
}
```

Try Statement

```
interface TryStatement {
    type: 'TryStatement';
    block: BlockStatement;
    handler: CatchClause | null;
    finalizer: BlockStatement | null;
}
```

with

```
interface CatchClause {
    type: 'CatchClause';
    param: Identifier | BindingPattern;
    body: BlockStatement;
}
```

Variable Declaration

```
interface VariableDeclaration {
    type: 'VariableDeclaration';
    declarations: VariableDeclarator[];
    kind: 'var' | 'const' | 'let';
}
```

with

```
interface VariableDeclarator {
    type: 'VariableDeclarator';
    id: Identifier | BindingPattern;
    init: Expression | null;
}
```

While Statement

```
interface WhileStatement {
    type: 'WhileStatement';
    test: Expression;
    body: Statement;
}
```

With Statement

```
interface WithStatement {
    type: 'WithStatement';
    object: Expression;
    body: Statement;
}
```

Scripts and Modules

A program can be either a script or a module.

```
interface Program {
  type: 'Program';
  sourceType: 'script';
  body: StatementListItem[];
}

interface Program {
  type: 'Program';
  sourceType: 'module';
  body: ModuleItem[];
}
```

with

```
type StatementListItem = Declaration | Statement;
type ModuleItem = ImportDeclaration | ExportDeclaration | StatementListItem;
```

Import Declaration

```
type ImportDeclaration {
    type: 'ImportDeclaration';
    specifiers: ImportSpecifier[];
    source: Literal;
}
```

with

```
interface ImportSpecifier {
    type: 'ImportSpecifier' | 'ImportDefaultSpecifier' | 'ImportNamespaceSpecifier';
    local: Identifier;
    imported?: Identifier;
}
```

Export Declaration

An export declaration can be in the form of a batch, a default, or a named declaration.

```
type ExportDeclaration = ExportAllDeclaration | ExportDefaultDeclaration |␣
↪ExportNamedDeclaration;
```

Each possible export declaration is described as follows:

```
interface ExportAllDeclaration {
    type: 'ExportAllDeclaration';
    source: Literal;
}

interface ExportDefaultDeclaration {
    type: 'ExportDefaultDeclaration';
    declaration: Identifier | BindingPattern | ClassDeclaration | Expression |␣
↪FunctionDeclaration;
}

interface ExportNamedDeclaration {
    type: 'ExportNamedDeclaration';
    declaration: ClassDeclaration | FunctionDeclaration | VariableDeclaration;
    specifiers: ExportSpecifier[];
    source: Literal;
}
```

with

```
interface ExportSpecifier {
    type: 'ExportSpecifier';
    exported: Identifier;
    local: Identifier;
};
```