WebIDL 2

npm package 24.2.2

Purpose

This is a parser for the WebIDL language. If you don't know what that is, then you probably don't need it. It is meant to be used both in Node and in the browser (the parser likely works in other JS environments, but not the test suite).

Installation

Just the usual. For Node:

```
npm install webidl2
```

In the browser:

```
<script src='webidl2.js'></script>
```

Documentation

The API to WebIDL2 is trivial: you parse a string of WebIDL and it returns a syntax tree.

Parsing

In Node, that happens with:

```
var WebIDL2 = require("webidl2");
var tree = WebIDL2.parse("string of WebIDL");
```

In the browser:

```
<script src='webidl2.js'></script>
<script>
  var tree = WebIDL2.parse("string of WebIDL");
</script>
```

Errors

When there is a syntax error in the WebIDL, it throws an exception object with the following properties:

- `message` : the error message
- `line` : the line at which the error occurred.
- `input` : a short peek at the text at the point where the error happened
- `tokens` : the five tokens at the point of error, as understood by the tokeniser (this is the same content as `input` , but seen from the tokeniser's point of view)

The exception also has a `toString()` method that hopefully should produce a decent error message.

AST (Abstract Syntax Tree)

The `parse()` method returns a tree object representing the parse tree of the IDL. Comment and white space are not represented in the AST.

The root of this object is always an array of definitions (where definitions are any of interfaces, dictionaries, callbacks, etc. — anything that can occur at the root of the IDL).

IDL Type

This structure is used in many other places (operation return types, argument types, etc.). It captures a WebIDL type with a number of options. Types look like this and are typically attached to a field called `idlType`:

```
{
  "type": "attribute-type",
  "generic": null,
  "idlType": "unsigned short",
  "nullable": false,
  "union": false,
  "extAttrs": [...]
}
```

Where the fields are as follows:

- `type`: String indicating where this type is used. Can be `null` if not applicable.
- `generic`: String indicating the generic type (e.g. "Promise", "sequence"). `null` otherwise.
- `idlType`: Can be different things depending on context. In most cases, this will just be a string with the type name. But the reason this field isn't called "typeName" is because it can take more complex values. If the type is a union, then this contains an array of the types it unites. If it is a generic type, it contains the IDL type description for the type in the sequence, the eventual value of the promise, etc.
- `nullable`: Boolean indicating whether this is nullable or not.
- `union`: Boolean indicating whether this is a union type or not.
- `extAttrs`: A list of extended attributes.

Interface

Interfaces look like this:

```
{
  "type": "interface",
  "name": "Animal",
  "partial": false,
  "members": [...],
  "inheritance": null,
  "extAttrs": [...]
}, {
  "type": "interface",
  "name": "Human",
  "partial": false,
  "members": [...],
  "inheritance": "Animal",
  "extAttrs": [...]
}
```

The fields are as follows:

- `type` : Always "interface".
- `name` : The name of the interface.
- `partial` : A boolean indicating whether it's a partial interface.
- `members` : An array of interface members (attributes, operations, etc.). Empty if there are none.
- `inheritance` : A string giving the name of an interface this one inherits from, `null` otherwise. **NOTE**: In v1 this was an array, but multiple inheritance is no longer supported so this didn't make sense.
- `extAttrs` : A list of extended attributes.

Interface mixins

Interfaces mixins look like this:

```
{
  "type": "interface mixin",
  "name": "Animal",
  "partial": false,
  "members": [...],
  "extAttrs": [...]
}, {
  "type": "interface mixin",
  "name": "Human",
  "partial": false,
  "members": [...],
  "extAttrs": [...]
}
```

The fields are as follows:

- `type` : Always "interface mixin".
- `name` : The name of the interface mixin.
- `partial` : A boolean indicating whether it's a partial interface mixin.
- `members` : An array of interface members (attributes, operations, etc.). Empty if there are none.
- `extAttrs` : A list of extended attributes.

Namespace

Namespaces look like this:

```
{
  "type": "namespace",
  "name": "Console",
  "partial": false,
  "members": [...],
  "extAttrs": [...]
}
```

The fields are as follows:

- `type` : Always "namespace".
- `name` : The name of the namespace.
- `partial` : A boolean indicating whether it's a partial namespace.
- `members` : An array of namespace members (attributes and operations). Empty if there are none.
- `extAttrs` : A list of extended attributes.

Callback Interfaces

These are captured by the same structure as Interfaces except that their `type` field is "callback interface".

Callback

A callback looks like this:

```
{
  "type": "callback",
  "name": "AsyncOperationCallback",
  "idlType": {
    "type": "return-type",
    "sequence": false,
    "generic": null,
    "nullable": false,
    "union": false,
    "idlType": "void",
    "extAttrs": []
  },
  "arguments": [...],
  "extAttrs": []
}
```

The fields are as follows:

- `type` : Always "callback".
- `name` : The name of the callback.
- `idlType` : An IDL Type describing what the callback returns.
- `arguments` : A list of arguments, as in function paramters.
- `extAttrs` : A list of extended attributes.

Dictionary

A dictionary looks like this:

```
{
  "type": "dictionary",
  "name": "PaintOptions",
  "partial": false,
  "members": [{
    "type": "field",
    "name": "fillPattern",
    "required": false,
    "idlType": {
      "type": "dictionary-type",
      "sequence": false,
      "generic": null,
      "nullable": true,
      "union": false,
      "idlType": "DOMString",
      "extAttrs": [...]
    },
    "extAttrs": [],
    "default": {
      "type": "string",
      "value": "black"
    }
  }],
  "inheritance": null,
  "extAttrs": []
}
```

The fields are as follows:

- `type` : Always "dictionary".
- `name` : The dictionary name.
- `partial` : Boolean indicating whether it's a partial dictionary.
- `members` : An array of members (see below).
- `inheritance` : A string indicating which dictionary is being inherited from, `null` otherwise.
- `extAttrs` : A list of extended attributes.

All the members are fields as follows:

- `type` : Always "field".
- `name` : The name of the field.
- `required` : Boolean indicating whether this is a required field.
- `idlType` : An IDL Type describing what field's type.
- `extAttrs` : A list of extended attributes.
- `default` : A default value, absent if there is none.

Enum

An enum looks like this:

```
{
  "type": "enum",
  "name": "MealType",
  "values": [
    { "type": "string", "value": "rice" },
    { "type": "string", "value": "noodles" },
    { "type": "string", "value": "other" }
  ],
  "extAttrs": []
}
```

The fields are as follows:

- `type` : Always "enum".
- `name` : The enum's name.
- `values` : An array of values.
- `extAttrs` : A list of extended attributes.

Typedef

A typedef looks like this:

```
{
  "type": "typedef",
  "idlType": {
    "type": "typedef-type",
    "sequence": true,
    "generic": "sequence",
    "nullable": false,
    "union": false,
    "idlType": {
      "type": "typedef-type",
      "sequence": false,
      "generic": null,
      "nullable": false,
      "union": false,
      "idlType": "Point",
      "extAttrs": [...]
    },
    "extAttrs": [...]
  },
  "name": "PointSequence",
  "extAttrs": []
}
```

The fields are as follows:

- `type` : Always "typedef".
- `name` : The typedef's name.
- `idlType` : An IDL Type describing what typedef's type.
- `extAttrs` : A list of extended attributes.

Implements

An implements definition looks like this:

```
{
  "type": "implements",
  "target": "Node",
  "implements": "EventTarget",
  "extAttrs": []
}
```

The fields are as follows:

- `type` : Always "implements".
- `target` : The interface that implements another.
- `implements` : The interface that is being implemented by the target.
- `extAttrs` : A list of extended attributes.

Includes

An includes definition looks like this:

```
{
  "type": "includes",
  "target": "Node",
  "includes": "EventTarget",
  "extAttrs": []
}
```

The fields are as follows:

- `type` : Always "includes".
- `target` : The interface that includes an interface mixin.
- `includes` : The interface mixin that is being included by the target.
- `extAttrs` : A list of extended attributes.

Operation Member

An operation looks like this:

```
{
  "type": "operation",
  "getter": false,
  "setter": false,
  "deleter": false,
  "static": false,
  "stringifier": false,
  "idlType": {
    "type": "return-type",
    "sequence": false,
    "generic": null,
    "nullable": false,
    "union": false,
    "idlType": "void",
    "extAttrs": []
  },
  "name": "intersection",
  "arguments": [{
    "optional": false,
    "variadic": true,
    "extAttrs": [],
    "idlType": {
      "type": "argument-type",
      "sequence": false,
      "generic": null,
      "nullable": false,
      "union": false,
      "idlType": "long",
      "extAttrs": [...]
    },
    "name": "ints"
  }],
  "extAttrs": []
}
```

The fields are as follows:

- `type` : Always "operation".
- `getter` : True if a getter operation.
- `setter` : True if a setter operation.
- `deleter` : True if a deleter operation.
- `static` : True if a static operation.
- `stringifier` : True if a stringifier operation.
- `idlType` : An IDL Type of what the operation returns. If a stringifier, may be absent.
- `name` : The name of the operation. If a stringifier, may be `null` .
- `arguments` : An array of arguments for the operation.
- `extAttrs` : A list of extended attributes.

Attribute Member

An attribute member looks like this:

```
{
  "type": "attribute",
  "static": false,
  "stringifier": false,
  "inherit": false,
  "readonly": false,
  "idlType": {
    "type": "attribute-type",
    "sequence": false,
    "generic": null,
    "nullable": false,
    "union": false,
    "idlType": "RegExp",
    "extAttrs": [...]
  },
  "name": "regexp",
  "extAttrs": []
}
```

The fields are as follows:

- `type` : Always "attribute".

- `name` : The attribute's name.
- `static` : True if it's a static attribute.
- `stringifier` : True if it's a stringifier attribute.
- `inherit` : True if it's an inherit attribute.
- `readonly` : True if it's a read-only attribute.
- `idlType` : An IDL Type for the attribute.
- `extAttrs` : A list of extended attributes.

Constant Member

A constant member looks like this:

```
{
  "type": "const",
  "nullable": false,
  "idlType": {
    "type": "const-type",
    "sequence": false,
    "generic": null,
    "nullable": false,
    "union": false,
    "idlType": "boolean"
    "extAttrs": []
  },
  "name": "DEBUG",
  "value": {
    "type": "boolean",
    "value": false
  },
  "extAttrs": []
}
```

The fields are as follows:

- `type` : Always "const".
- `nullable` : Whether its type is nullable.
- `idlType` : An IDL Type of the constant that represents a simple type, the type name.
- `name` : The name of the constant.
- `value` : The constant value as described by Const Values
- `extAttrs` : A list of extended attributes.

Arguments

The arguments (e.g. for an operation) look like this:

```
{
  "arguments": [{
    "optional": false,
    "variadic": true,
    "extAttrs": [],
    "idlType": {
      "type": "argument-type",
      "sequence": false,
      "generic": null,
      "nullable": false,
      "union": false,
      "idlType": "long",
      "extAttrs": [...]
    },
    "name": "ints"
  }]
}
```

The fields are as follows:

- `optional` : True if the argument is optional.
- `variadic` : True if the argument is variadic.
- `idlType` : An IDL Type describing the type of the argument.
- `name` : The argument's name.
- `extAttrs` : A list of extended attributes.

Extended Attributes

Extended attributes are arrays of items that look like this:

```
{
  "extAttrs": [{
    "name": "TreatNullAs",
    "arguments": null,
    "type": "extended-attribute",
    "rhs": {
      "type": "identifier",
      "value": "EmptyString"
    }
  }]
}
```

The fields are as follows:

- `name` : The extended attribute's name.
- `arguments` : If the extended attribute takes arguments (e.g. `[Foo()]`) or if its right-hand side does (e.g. `[NamedConstructor=Name(DOMString blah)]`) they are listed here. Note that an empty arguments list will produce an empty array, whereas the lack thereof will yield a `null` . If there is an `rhs` field then they are the right-hand side's arguments, otherwise they apply to the extended attribute directly.
- `type` : Always `"extended-attribute"` .
- `rhs` : If there is a right-hand side, this will capture its `type` (which can be "identifier" or "identifier-list") and its `value` .

Default and Const Values

Dictionary fields and operation arguments can take default values, and constants take values, all of which have the following fields:

- `type` : One of string, number, boolean, null, Infinity, NaN, or sequence.

For string, number, boolean, and sequence:

- `value` : The value of the given type, as a string. For sequence, the only possible value is `[]` .

For Infinity:

- `negative` : Boolean indicating whether this is negative Infinity or not.

`iterable<>` , `legacyiterable<>` , `maplike<>` , `setlike<>` declarations

These appear as members of interfaces that look like this:

```
{
  "type": "maplike", // or "legacyiterable" / "iterable" / "setlike"
  "idlType": /* One or two types */ ,
  "readonly": false, // only for maplike and setlike
  "extAttrs": []
}
```

The fields are as follows:

- `type` : Always one of "iterable", "legacyiterable", "maplike" or "setlike".
- `idlType` : An array with one or more IDL Types representing the declared type arguments.
- `readonly` : Whether the maplike or setlike is declared as read only.
- `extAttrs` : A list of extended attributes.

Testing

Running

The test runs with mocha and expect.js. Normally, running mocha in the root directory should be enough once you're set up.

Coverage

Current test coverage, as documented in `coverage.html` , is 95%. You can run your own coverage analysis with:

```
jscoverage lib lib-cov
```

That will create the lib-cov directory with instrumented code; the test suite knows to use that if needed. You can then run the tests with:

```
JSCOV=1 mocha --reporter html-cov > coverage.html
```

Note that I've been getting weirdly overescaped results from the html-cov reporter, so you might wish to try this instead:

```
JSCOV=1 mocha  --reporter html-cov | sed "s/&lt;/</g" | sed "s/&gt;/>/g" | sed
"s/&quot;/\"/g" > coverage.html
```

Browser tests

In order to test in the browser, get inside `test/web` and run `make-web-tests.js`. This will generate a `browser-tests.html` file that you can open in a browser. As of this writing tests pass in the latest Firefox, Chrome, Opera, and Safari. Testing on IE and older versions will happen progressively.