DATE: 11 February 2009

DOC TYPE: PDF file

TITLE: Module System for ES-Harmony

SOURCE [1]: Mr. Kris Kowal (FastSoft Inc.) and Mr. Ihab Awad (Google)

STATUS: draft technical contribution

ACTION ID: FYI

NO. OF PAGES:

[1] *NOTE*
The source of this document grants permission to Ecma International to publish this document internally and / or externally (at least one option must be selected)

Introduction

This is a proposal for a module system for inclusion in ES-Harmony and possible prototyping prior to standardization.

This proposal does not introduce backward-incompatible changes to ES. We specify semantics and syntax for constructing secure sandboxes for a new kind of managed script: a module, that shares most of the semantics of a conventional ES script, but does not allow access or assignment to free variables, has an implicit function block scope that contains a reference to the sandbox interface and a reference to the object intended to be exported to other modules, and does not have access to a global scope. We leave the means of invoking such systems as an exercise for ES engine implementors. Sandboxes have singleton module instances and shared capabilities, provided by the sandbox constructor. We specify the interface of a sandbox but not its implementation. Capability objects are the only means by which a module can communicate with objects outside of the sandbox. We specify that the architecture of a module loader divides identifying, fetching, evaluating, and constructing module instances into separate layers that can each be cached, memoized, and securely shared. We specify a means by which sandboxes can be created within sandboxes so that capabilities can be restricted. Modules cannot modify each other's exported objects (they are frozen) and modules are sovereign over their own namespace (with the exception of the names "exports" and "require") and to that end do not have any objects on their scope chain that are shared with either modules, sandboxes, or global scripts. Module evaluation supports "import" and "export" statements that use the "`require`" and "`exports`" objects, providing a backward compatibility layer so that analogous but insecure systems can be implemented with pre-Harmony syntax.

Roadmap

Module loader syntax would support migration in three phases from traditional ES global scripts, to modules that use special functions to import each other, then finally to a concise syntax for importing. Both of these latter syntacies would support absolute and relative module identifiers, destructured imports (selection and renaming), and sandboxing.

(Legacy) Global Script Module System: a module loader that can be implemented with currently deployed versions of ES, using only capabilities provided by their respective environments (`eval`, `with`, XHR, local file access). A legacy module loader could support a common, transitional coding style that does not require additional features in the ES syntax. Global script module systems can not guarantee capability isolation without the assistance of an external translator or validator like Caja, ADsafe, or Jacaranda because all modules would share a mutable transitive global object that cannot be removed from the scope chain.

(Future, Native) ECMAScript Module System: a module loader that accepts the transitional syntax, but also provides syntactic sugar for imports and exports, and guarantees that groups of modules can only access certain capabilities, and can limit the capabilities for a subordinate group of modules to which they do not wish to be vulnerable. Future module loaders would not distinguish nor forbid mixing of transitional syntax with syntactic sugar. Environments that provide a native module system could supplant the legacy module system so that the client can remain agnostic of whether their module loader is legacy or native.

Module formats

Module syntax has several aspects that this document will explain in layers:

- importing capabilities from a sandbox,
- exporting a module,
- importing a module,
- importing a module asynchronously, and
- creating a sandbox with restricted capabilities.

At each layer, we'll explore three kinds of syntax:

- syntactic sugar,
- so called "syntactic salt", and
- a pedagogical desugared syntax.

We define these three kinds of syntax:

Sugar (syntactic sugar): syntax in modules that requires modifications to the ES grammar. Sugary syntax will invoke equivalent behavior of corresponding "salty" syntax.

Salt (syntactic salt): transitional syntax that provides module features without requiring support for new syntax: no parsing, no analysis, just function application and object references. Both legacy and native module loaders would accept syntactic salt.

Desugared: syntax that is equivalent to corresponding syntactic sugar by behavior, that a user might not be able to write, that an interpreter might use as an intermediate step, but is otherwise rendered to help explain how the sugary syntax works. Desugared syntax is nearly identical to syntactic salt, except that it wraps the salty syntax in a module constructor function envelope and assumes that future ES features are available.

We define the following module formats accessible to the programmer:

Salty module: a chunk of ES text crafted to run in either a global-script-module-system or a future-ecmascript-module-system.

Sugary module: a chunk of ES text that has syntactic sugar that can only be evaluated by a future-ecmascript-module-system.

We also define for contrast:

Global script: a chunk of ES text crafted to be evaluated in global context like ECMAScript today.

Importing capabilities and exporting a module

Desugared module

Given the definition:

> **Closed function:** A function that has no free variables.

We assume that the desugared form of any ES module constructor is a closed function that takes exactly two arguments, a frozen ES module importer containing capabilities, and an object into which module exports are assigned, for example:

```
function (require, exports) {

  const {document, bgColor} = require.env;

  var isOn = false;
```

```
        let toggle = exports.toggle = function () {
          (isOn ? reset : set)();
        };
        let set = exports.set = function () {
          document.setBackgroundColor(color);
          isOn = true;
        };
        let reset = exports.reset = function () {
          document.setBackgroundColor('#ffffff');
          isOn = false;
        };

      }
```

This proposal does not require nor forbid that the interpreter statically verify that there are no free variables in a module, but does require that any accessed or assigned free variables raise a `NameError` at the point of execution.

Sugary module

The above module would be desugared from the syntax, only compatible with ES-Harmony implementations, as follows:

```
const {document, bgColor} = require.env;

var isOn = false;

export toggle = function () {
  (isOn ? reset : set)();
};
export set = function () {
  document.setBackgroundColor(background);
  isOn = true;
};
export reset = function () {
  document.setBackgroundColor('#ffffff');
  isOn = false;
};
```

The desugaring algorithm is to expand the statements of the form:

```
export <identifier> = <value>;
```

then wrap the resulting text in the envelope:

```
( function (require, exports) { <desugared> } )
```

Note that there is no special syntax for assigning the module capabilities (obtained from "`require.env`") to local variables, since ES-Harmony destructuring syntax already makes this easy enough.

Salty module

We would support an intermediate syntax that would permit developers to migrate from global scripts to modules that support all of the module loader features using a module loader system implemented with current ECMAScript tools. This system would not provide security guarantees, but would allow some scripts to be used in both environments. We show this syntax in ES3.

```
var document = require.env.document;
var bgColor = require.env.bgColor;

var isOn = false;

exports.toggle = function () {
    (isOn ? reset : set)();
};
exports.set = function () {
    document.setBackgroundColor(color);
    isOn = true;
};
exports.reset = function () {
    document.setBackgroundColor('#ffffff');
    isOn = false;
};
```

The assignments to `exports` add the symbols to the local scope (e.g. note their use inside the exported function `toggle`). This is brought about since module systems wishing to support this format must use a module function envelope of the following form:

```
(function (require, exports) {
  with (exports) {
    (function () {
      <desugared>
    }).call(this);
  }
})
```

Global / Salty Script / Module

Programmers can write global ES scripts in such a way that they continue to be used as global modules. Instead of assigning to "`exports`" (the *modus operandi* for salty scripts hitherto described) we assign our exportables to "`this`". We also wrap the file in a closure. The closure is redundant with the module constructor closure if you're using the file as a module, but it ensures module pattern when the file is used as a global script. To that end, if we alter the sandbox such that the module constructor function receives its "`exports`" object as "`this`", we can write scripts that can also be used as modules.

```
(function () {

  var document = typeof require == "undefined" ? document :
require.env.document;
  var bgColor = typeof require == "undefined" ? bgColor :
require.env.bgColor;

  var isOn = false;

  var toggle = this.toggle = function () {
    (isOn ? reset : set)();
  };
  var set = this.set = function () {
    document.setBackgroundColor(color);
    isOn = true;
  };
  var reset = this.reset = function () {
    document.setBackgroundColor('#ffffff');
    isOn = false;
```

```
      };

    )).call(this);
```

Calling the enclosure with "`call(this)`" is necessary to guarantee that "`this`" is bound to "`exports`" (in a module) or "`global`" (in a global script) both inside and outside the closure. This is brought about since module systems wishing to support this format must execute the module function in a block of the following form:

```
moduleFunction.call(exports, require, exports);
```

Global and salty support

Support for global and salty modules requires a certain amount of concession to the ideal of lexical scoping. We consider this acceptable since (a) they are backward compatibility shims; (b) module systems may choose not to support them; and (c) they do not intrude on the source format of modules that do not wish to use these features.

Module loaders

We define:

Module text: The ES program text of a module.

Module function (or constructor): The ES function value that is produced by compiling the module text.

Module identifier: A value that identifies a module, nominally referring to a file that contains its text.

Module fetcher: A function that maps module identifiers to module text.

Module evaluator: A function that maps module text to a module constructor function. This function is also responsible for desugaring the module text and checking it for correctness.

Module loader: A function that maps module identifiers to module constructors. As such, a module loader encapsulates a fetcher and evaluator and may cache (not necessarily memoizing) either or both module texts and module constructors.

Module loaders are installation-dependent. For example, the Web browser environment may provide a module loader which accepts module identifiers in the form of URIs to the module text:

```
"file:///usr/lib/js/util/linkedList.js"
"http://example.com/myModule.js"
```

Another loader may require a module identifier consisting of a dotted name and a version range:

```
{ name: "com.example.util.linkedList", version: { min: "3.70", max:
"4.00" } }
```

Yet another may allow path names relative to the importing module:

```
"../util/sortAlgorithms.js"
```

And another module loader might use a dotted name notation:

```
    ".util.sortAlgorithms"
```

Since it would be trivial to construct an adapter for any of these notations and supply that to a child sandbox, we feel no need to constrain implementations to a particular module identifier syntax.

So that branches of a module identifier name space can be reorganized without modifying the module texts in that branch, it's desirable for module loaders to support a notion of self-relative module imports. That is, if you're looking at the text of a module with a given identifier, and the identifiers are hierarchical, load and require could accept a module identifier with a given prefix that indicates that it ought to be resolved relative to the importing module's identifier. Consider this notation where module identifiers are local file system paths:

```
    // in "/usr/lib/ecmascript/framework/extension.es
    import "./base.es" as base
```

The identifier "`./base.es`" would get resolved relative to "`/usr/lib/ecmascript/framework/extension.es`" to "`/usr/lib/ecmascript/framework/base.es`". Presumably, the identifier "`base.es`" would have resolved to "`/usr/lib/ecmascript/base.es`" instead, off the base directory for fully qualified module identifiers, "`/usr/lib/ecmascript/`". A module fetcher could opt to do analogous resolutions with URL's or dot delimited names.

A loader is an object required to have the following interface:

- `resolve`: takes a partially or fully qualified identifier and an optional fully qualified identifer, and returns the former fully qualified against the latter. "`resolve`" throws an exception if the first identifier is partially qualified and the base identifier is not passed or is `undefined`.
- `fetch`: takes a partially or fully qualified identifier and an optional fully qualified base identifier, and returns the corresponding module text.
- `evaluate`: takes a module text and returns a constructor function.
- `load`: takes a partially or fully qualified identifier and an optional fully qualified base identifier, and returns a module constructor function, using "`resolve`", "`fetch`", and "`evaluate`".

Module sandboxes

Module capabilities: An object that contains capability bearing objects that would be available to any module in a given sandboxed environment via the `require.env` object. Capabilities would include objects that facilitate secure input and output, or as in a web page, the `window` and `document`.

Module instance: The value returned from the invocation of a module constructor when applied to a particular sandbox. A module instance is an `Object`, copied and frozen from a module's exports `Object`. Copying and freezing is deferred to the time of a request for the module instance so that a module can continue to build its `exports` object through to the completion of the module constructor function, even if there is a cyclic dependency chain that includes the module.

Module sandbox: A system of module instances with common capabilities. A sandbox has one inherent `require` function that is synonymous with the sandbox. The `require` function uses a given module loader to obtain module constructors which it executes to construct module instances with the sandbox capabilities. Sandboxes can be built inside parent sandboxes using whatever capabilities are granted to the parent sandbox. A sandbox must:

- be a function that accepts module identifiers and returns module instances.
- be memoized such that it appears stateless, that every call with a given identifier returns the same value.
- have a member called "`loader`" that is a loader: a function that accepts a module identifier and returns a module constructor.
- have a member "`env`" that is an object that contains zero or more capability laden objects.

A module sandbox provides a function that enables modules to import other modules in the sandbox by their identifier. If that module has never been imported by the sandbox before, the sandbox loads the module, instantiates it by calling its module function (providing the sandbox's module scope), and memoizes and returns the resulting module instance. Otherwise, the sandbox returns the previously memoized module instance.

Given a loader and an ES object containing bindings to be introduced into a module scope, any code may construct a sandbox:

```
var aLoader = /* ... */;
var aDocument = /* ... */;
var aColor = /* ... */;

var sandbox = Sandbox(aLoader, {document: aDocument, bgColor:
aColor});
```

Sandbox constructors are not privileged. The following is a valid implementation:

```
function Sandbox(loader, env) {
  env = copy(env).freeze();
  modules = Map();

  var require = function (id, relativeId) {
    if (!modules.has(id)) {
      var constructor = loader.load(id, relativeId);
      var exports = {};
      constructor.call(exports, require.relativeTo(id), exports);
      exports.freeze();
      modules.set(id, exports);
    }
    return modules.get(id);
  };

  require.relativeTo = function (baseId) {
    var requireRelative = function (relativeId) {
      return myRequire(relativeId, baseId);
    };
    requireRelative.id = baseId;
    requireRelative.env = env;
    requireRelative.loader = loader;
    return requireRelative.freeze();
  };

  require.env = env;
  require.load = load;

  return require.freeze();
}
```

All forms of module importing syntax ultimately get modules from a "`require`" function. A "`require`" object is not constrained to providing only "`loader`", "`env`" and "`id`", but we do not specify other attributes here.

Importing modules

As with the module format, there are three forms of importing a module: the desugared form; the syntactic sugar we propose for ES-Harmony, and the "salty" form for backward compatibility. Imagine that we wish to use a module with the sugared form:

```
const {document} = require.env;

export translateTo = function (language) {
  if (language === 'french') {
    document.write('Traduit en Francais!');
  } else if (language === 'english') {
    document.write('Translated to English!');
  }
};
```

and imagine that this module is stored at the URI:

```
"http://example.com/docTranslator.js"
```

Desugared module usage

The desugared form would use the `require` object (a function) to get an instance of a required module:

```
function (require, exports) {
  const document = require.env.document;
  const bgColor = require.env.bgColor;

  const {translateTo} = require
('http://example.com/docTranslator.js');

  var isOn = false;

  const toggle = exports.toggle = function () {
    (isOn ? reset : set)();
  };
  const set = exports.toggle = function () {
    document.setBackgroundColor(color);
    isOn = true;
  };
  const reset = exports.reset = function () {
    document.setBackgroundColor('#ffffff');
    isOn = false;
  };
  const translate = exports.translate = function() {
    translateTo("french");
  };

}
```

Salty module usage

The salty syntax version would be as follows:

```
var document = require.env.document;
var bgColor = require.env.bgColor;

var isOn = false;

var translateTo = require
('http://example.com/docTranslator.js').translateTo;

exports.toggle = function () {
  (isOn ? reset : set)();
};
exports.set = function () {
    document.setBackgroundColor(color);
    isOn = true;
};
exports.reset = function () {
  document.setBackgroundColor('#ffffff');
  isOn = false;
};
exports.translate = function() {
  translateTo("french");
};
```

Sugary module usage

The syntactically sugared form of this module would be as follows:

```
const {document, bgColor} = require.env;

from 'http://example.com/docTranslator.js' import translateTo;

var isOn = false;

export toggle = function () {
  (isOn ? reset : set)();
};
export set = function () {
  document.setBackgroundColor(background);
  isOn = true;
};
export reset = function () {
  document.setBackgroundColor('#ffffff');
  isOn = false;
};
export translate = function() {
  translateTo("french");
};
```

Module Import Syntax

We provision syntax for:

- importing and binding modules to local scope identifiers,
- importing selected exports from a module and binding them to local identifiers by the same name,
- importing selected exports from a module and binding them to local identifiers with reassigned names,
- importing from a given sandbox (with all previous forms).

In all of the following cases, X and Y are both expressions, which is to say that object literals and variables are both acceptable in these place holders.

Sugar	**Desugar**	**Salt**
`import X as Y;`	`const Y = require(X);`	`var Y = require(X);`

`import X as Y, W as Z;`	`const Y = require(X);`
	`const Z = require(W);`
		`var Y = require(X);`
		`var Z = require(W);`

`from X import A;`	`const {A} = require(X);`	`var A = require(X).A;`

`from X import A, B;`	`const {A, B} = require(X);`	`var A = require(X).A;`
		`var B = require(X).B;`

`from X import A as F;`	`const {A: "F"} = require(X);`	`var F = require(X).A;`
`import X as Y with S;`	`const Y = S(X);`	`var Y = S(X);`

There would be three forms in the grammar for importing, one for bringing module objects into the module scope, and two others for copying exports from a module into our own module scope.

```
"import" ((expression ["as" name]) % ",") ["with" expression] ";"
"from" expression "import" ((name ["as" name]) % ",") ["with"
expression] ";"
"from" expression "import" "*"
```

Reflexive Assignment Extension

A module can bind an imported module to a local variable name that is the same as its identifier. This requires the module identifier to be a string that is a valid chain of dot-delimited variable names. Any prefix variable names that do not exist would be constructed as empty objects. Dot prefixes would be ignored so that the module loader implementation can reserve dot prefixes for module relative imports.

Sugar	**Desugar**	**Salt**
`import "X";`	`const X = require("X");`	`var X = require("X");`
`import ".X";`	`const X = require("X");`	`var X = require("X");`

`import "X.Y";`	`let X = {};`
	`X.Y = require("Y");`
		`var X = {};`
		`X.Y = require("Y");`

Import All Extension

A module can adopt all of the exported variables of a given module, by the same names.

Sugar	**Desugar**	**Salt**
`from X import *`	`__update__(__locals__, require(X));`	`require.all(X);`
`from X import A, B`	`const {A, B} = require(X);`	`require.all(X, ["A", "B"]);`
`from X import A as F`	`const {F: "A"} = require(X);`	`require.all(X, {F: "A"});`

This extension to the import syntax imposes the additional constraint on a sandbox that it have an "all" function that has the ability to inject variables in the caller scope, so called "dynamic scope". I'm going to take a wild guess that this is not going to fly well past our security proponents or the "don't break lexical scope" crowd, so I include this section as a warning to myself and others that use and like this syntactic form. Maybe we can construct some unlikely kind of compromise.

However, as a byproduct of the inclusion of an "all" function to the sandbox, the salty syntax for destructuring imports in Pre-Harmony ES could be greatly improved. This feature would also have to be provided by a future ES module loader for compatibility.

Sugar	**Desugar**	**Salt**
`from X import A, B`	`const {A, B} = require(X);`	`require.all(X, ["A", "B"]);`
`from X import A as F`	`const {F: "A"} = require(X);`	`require.all(X, {F: "A"});`

Asynchronous Module Importing Extension

So far all syntax examples presume that a script will block until dependencies are resolved. This will no doubt be unacceptable in certain environments. We can resolve this issue by implying that the syntactic sugar can (at the purview of the ES engine implementor) desugar to a form of implicit continuation passing such that the main ES event queue, and related queues in the embedding system, can continue processing until the module instance is available. At that point, the continuation may be called back with the module instance and resume execution.

Syntactic Sugar

```
import X as Y
...
```

Desugared Syntax

The above syntax implies that any import directive

```
require(X, function (Y) {
    ...
});
```

Syntactic Salt

This transformation cannot be applied in pre-Harmony ES, and probably not even post-Harmony, if I'm not mistaken. For that reason, the desugarred syntax must be explicitly supported as salty syntax if module authors in previous versions of ES can take advantage of non-blocking scripts. This poses certain onerous constraints for sandbox implementors. For example, a sandbox must schedule module loading. That is, the sandbox must retain notes about whether all continuations, including the module constructor itself, have run to completion, before calling back to dependent modules. In salty syntax, mixing the continuation passing style and blocking import calls is not possible because an importing module cannot know whether the depended module will defer to a continuation.

Addendum: Bind / Curry Calling Module

code from the module relative imports extension is in peach.
code from the global/module extension is in yellow.
new code is in green.

```
;
          constructor.call(exports, require.relativeTo(id), exports);
          modules.set(id, exports);
        }

        // curry exports with the id if they request
        let module = modules.get(id);
        let boundModule = {};
        for (let name in module) {
          if (module.hasOwnProperty(name)) {
            if (module[name].curryCallerModuleId) {
              boundModule[name] = (function (unbound) {
                return unbound.apply(this, [relativeId] + arguments);
              })(module[name]);
            } else {
              boundModule[name] = module[name];
            }
          }
        };

        return boundModule.freeze();
      }

      require.relativeTo = function (relativeId) {
        var requireRelative = function (id) {
          return require(id, relativeId);
        };
        requireRelative.env = env;
        requireRelative.load = load;
        return requireRelative.freeze();
      };

      require.env = env;
      require.load = load;

      return require.freeze();
    }

    /* a decorator that is trivial to define or import inside the sandbox
    */
    export curryCallerModuleId = function (function) {
      function.curryCallerModuleId = true;
      return function;
    };

    /* assuming a module exists with a log function, presumably connected
    to a
       console capability object. */
    from "console" import log as baseLog;

    /* provide a log function that notes the module it was called in */
    export log = curryCallerModuleId(function (id, message) {
      baseLog(message + " from " + id);
    });
```

Addendum: Module Bundling

In environments where there is high latency in fetching, it's desirable to prefetch module constructors. There are various strategies.

- The client can perform static analysis of the module text or the abstract syntax of the module constructor to heuristically determine the module identifiers of modules that are likely to be required and issue requests for the corresponding files, recursively fetching their transitive dependencies.
- A sandbox can arrange to download an archive of module texts to prime its module text cache or memo. This requires agreement between the sandbox and server that certain modules must be available in an archive, or that a script archiving service is available.
- The server can analyze the module text or the abstract syntax of the module constructor and subvert requests for a module text with:
    - an archive containing the module text and the texts of its transitive dependencies, or
    - a module that registers module constructor functions for their transitive dependencies
- Use a build system to analyze the modules in a library and create a "production" version of the library that replaces module files with versions that are either archives or register constructor functions for their probable dependencies.

All of these options are possible with extensions to a loader or sandbox.

Addendum: Module metadata

To enforce clean isolation, our proposal does not allow modules to name themselves or their contents in their client's universe of discourse. However, it may be useful for the module function to support metadata. This would be visible after loading:

```
var aLoader = /* ... */;
var moduleFcn = aLoader.load("http://example.com/test.js");
m.version; // → '1.8.10b'
m.author;  // → 'Alyssa P Hacker <hacker@example.com>'
m.imports; // → [ 'document', 'background' ]
m.exports; // → [ 'toggleIt', 'setIt' ]
```

This may be assigned using standardized comment-like notations analogous to Javadoc:

```
/*file module.js */
/*version 1 */
/*author Ihab Awad, Kris Kowal */
/*quality dismal */
/*tutorial-order 1 */

/*preamble
    Copyright 2008 Ihab Awad, Kris Kowal
    We have a license!  You can read it at the bottom of this file.
*/

/**
    This is inline documentation.
*/

/*** Map
    This is documentation for the Map function.
*/
export function Map () {
}
```

Addendum: Implementing a Loader

Here's the rest of a trivial implementation (untested) of a sandbox. This bit of code builds a loader from a fetcher and a resolve function for the fetcher's module identifier name space. The only part of this code that cannot be implemented in legacy ECMAScript is the evaluator. We need an "`eval`" function that evaluates the program in a context that has frozen primordial objects (`Array`, `Object`, &c), has an undefined (or provided) global object, and an empty (or provided) scope chain. There must be no way for code executed in the exection environment to receive or share lexemes with other modules.

```
/* returns an absolute (fully-qualified) identifier
   for a given identifier (that may be absolute or relative)
   and a base identifier (that is optionally undefined).
   if the identifier is absolute, it just returns the id.
   if the identifier is not fully qualified, returns the equivalent
   absolute identifier from the given base identifier.
   raises an exception if the id is relative and
   the base id is undefined.
*/
let resolve = function (id, baseId) {...};

/* a decorator that returns a version of a function that
   only runs a computation once for a given set of arguments
   and stores the result for future requests */
let memoize = function (callback) {
  let memo = Map();
  return function () {
    if (!memo.has(arguments))
      memo.set(arguments, callback.apply(this, arguments));
    return memo.get(arguments);
  };
};

/* a decorator for functions that accept an optional relativeId */
let resolved = function (callback) {
  return function (id, relativeId) {
    return callback(resolve(id, relativeId));
  };
};

let fetch = resolved(memoize(...));

let evaluate = memoize(function (text) {
  if (supportSalt)
    text = "with(exports){(function(){" + text + "}).call(this)}";
  text =  "(function (require, exports) {" + text + "})";
  return hermeticEval(text);
});

let load = resolved(memoize(function (id) {
  return evaluate(fetch(id));
}));
```