Sweet.js Documentation

1 Introduction

Install the sweet.js compiler via npm:

```
npm install -g sweet.js
```

This installs the `sjs` binary which is used to compile sweet.js code:

```
sjs --output out.js my_sweet_code.js
```

2 Rule Macros

You can think of macros as functions that work on syntax. Much like a normal function you write a macro *definition* and then later *invoke* the macro with a syntax argument to produce new syntax. Running sweet.js code through the compiler will *expand* all macros and produce pure JavaScript that can be run in any JS environment.

Sweet.js provides two ways to define a macro: the simpler pattern-based *rule* macros and the more powerful procedural *case* macros (if you are familiar with Scheme or Racket these correspond to `syntax-rules` and `syntax-case`).

Rule macros work by matching a syntax *pattern* and generating new syntax based on a *template*.

To define a macro sweet.js provides a new `macro` keyword that looks something like this:

```
macro <name> {
  rule { <pattern> } => { <template> }
}
```

For example, here's a really simple identity macro that just expands to its syntax argument:

```
macro id {
  rule {
    // after the macro name, match:
    // (1) a open paren
    // (2) a single token and bind it to `$x`
    // (3) a close paren
    ($x)
  } => {
    // just return the token that is bound to `$x`
    $x
  }
}
```

We can then invoke `id` with:

```
id (42)
// expands to:
// 42
```

A pattern that begins with `$` matches any token and binds it to that name in the template while everything else matches literally.

Note that a single token includes matched delimiters not just numbers and identifiers. For example, an array with all of its elements counts as one token:

```
id ([1, 2, 3])
// expands to:
// [1, 2, 3]
```

Macros can have multiple rules.

```
macro m {
  rule { ($x) } => { $x }
```

```
  rule { ($x, $y) } => { [$x, $y] }
}

m (1);
m (1, 2);
```

Rules are matched in-order and the first one to match the pattern will be used.

And macros can be recursively defined:

```
macro m {
  rule { ($base) } => { [$base] }
  rule { ($head $tail ...) } => { [$head, m ($tail ...)] }
}
m (1 2 3 4 5)  // --> [1, [2, [3, [4, [5]]]]]
```

2.1 Patterns

2.1.1 Repetition

Repeated tokens can be matched with ellipses `...`.

```
macro m {
  rule { ($x ...) } => {
    // ...
  }
}
m (1 2 3 4)
```

A repeated pattern with a separator between each item can be matched by adding `(,)` between `...` and the pattern being repeated.

```
macro m {
  rule { ($x (,) ...) } => {
    [$x (,) ...]
  }
}
m (1, 2, 3, 4)
```

Repeated groups of patterns can be matched using `$()`.

```
macro m {
  rule { ( $($id = $val) (,) ...) } => {
    $(var $id = $val;) ...
  }
}
m (x = 10, y = 2)
```

2.1.2 Literal patterns

The syntax `$[]` will match what is inside the brackets literally. For example, if you need to match `...` in a pattern (rather than have `...` mean repetition) you can escape it with `$[...]`.

2.1.3 Named patterns

You can name pattern groups and literal groups. Sub-bindings can be referenced by concatenating the group name and binding name. All the syntax matched by the group will be bound to the group name.

```
macro m {
  rule { ($binding:($id = $val) (,) ...) } => {
    $(var $binding;) ...
    // Or with sub-bindings
    $(var $binding$id = $binding$val;) ...
  }
}

macro m {
  rule { $ellipsis:[...] } => {
    // ...
  }
}
```

2.1.4 Pattern Classes

A pattern name can be restricted to a particular parse class by using

`$name:class` in which case rather than matching a token the pattern matches all the tokens matched by the class.

```
macro m {
  rule { ($x:expr) } => {
    $x
  }
}
m (2 + 5 * 10)
// --> expands to
2 + 5 * 10
```

The pattern classes are:

- `:ident` -- matches an identifier (eg. `foo`)
- `:lit` -- matches a literal (eg. `100` or `"a string"`)
- `:expr` -- matches an expression (eg. `foo("a string") + 100`)

2.1.5 Custom Pattern Classes (Experimental)

Note that the syntax of custom pattern classes is currently experimental and subject to change.

You can define your own custom pattern classes with `macroclass`.

```
// define the cond_clause pattern class
macroclass cond_clause {
  pattern {
    rule { $check:expr => $body:expr }
  }
}

macro cond {
  rule { $first:cond_clause $rest:cond_clause ... } => {
    // sub-pattern variables in the custom class are
    // referenced by concatenation
    if ($first$check) {
      $first$body
    } $(else if ($rest$check) {
```

```
      $rest$body
    }) ...
  }
}

cond
  x < 3  => console.log("less than 3")
  x == 3 => console.log("3")
  x > 3  => console.log("greater than 3")
// expands to:
// if (x < 3) {
//     console.log('less than 3');
// } else if (x == 3) {
//     console.log('3');
// } else if (x > 3) {
//     console.log('greater than 3');
// }
```

You can also define multiple patterns and bind missing sub-pattern variables using `with`. You may supply multiple `with` declarations or separate bindings by commas.

```
macroclass alias_pair {
  pattern {
    rule { $from:ident as $to:ident }
  }
  pattern {
    rule { $from:ident }
    with $to = #{ $from };
  }
}

macro import {
  rule { { $import:alias_pair (,) ... } from $mod:lit ;... } => {
    var __module = require($mod);
    $(var $import$to = __module.$import$from;) ...
  }
  rule { $default:ident from $mod:lit ;... } => {
    var $default = require($mod).default;
  }
}

import { a, b as c, d } from 'foo'
// expands to:
```

```
// var __module = require('foo');
// var a = __module.a;
// var c = __module.b;
// var d = __module.d;
```

Patterns with only a `rule` declaration may be collapsed into just a `rule` declaration in lieu of a `pattern`.

2.1.6 Repeating variables in patterns

In contrast to other macro systems, sweet.js allows variables to appear more than once in the pattern. Similarily to repeated variables in the body of a rule macro, the syntax captured by a variable must be the same for all occurences of that variable, otherwise the macros will not match.

```
macro m {
    rule { $x $x } => { "the same!" }
    rule { $x $y } => { "different" }
}
m 1 1   // expands to 'the same!'
m 1 2   // expands to 'different'
```

Of course, this also applies to repeated variables in repitition groups which enables complex pattern matching that would otherwise require complex case macros.

```
macro m {
    rule { $x $($p:(+) $x) ... }
      => { $x * (1 $($p 1) ...) }
}
m 23 + 23 + 23   // expands to 23 * (1 + 1 + 1);
m 23 + 23 + 42   // expands to 23 * (1 + 1) + 42;
```

3 Hygiene

The most important property of sweet.js is hygiene. Hygiene prevents

variables names inside of macros from clashing with variables in the surrounding code. It's what gives macros the power to actually be syntactic abstractions by hiding implementation details and allowing you to use a hygienic macro *anywhere* in your code.

Hygiene protects against two kinds of naming collisions: binding collisions and reference collisions.

3.1 Hygiene Part 1 (Binding)

The part of hygiene most people intuitively grok is keeping track of the variable *bindings* that a macro introduces. For example, the swap macro creates a `tmp` variable that should only be bound inside of the macro:

```
macro swap {
  rule { ($a, $b) } => {
    var tmp = $a;
    $a = $b;
    $b = tmp;
  }
}

var tmp = 10;
var b = 20;
swap (tmp, b)
```

Without hygiene the two `tmp` variables would clash:

```
var tmp = 10;
var b = 20;

var tmp = tmp;
tmp = b;
b = tmp;
```

Hygiene keeps the two `tmp` bindings distinct by renaming them both:

```
var tmp$1 = 10;
```

```
var b = 20;

var tmp$2 = tmp$1;
tmp$1 = b;
b = tmp$2;
```

3.2 Hygiene Part 2 (Reference)

Hygiene also handles variable *references*. The body of a macro can contain references to bindings declared outside of the macro and those references must be consistent no matter the context in which the macro is invoked.

Some code to clarify. Let's say you have a macro that uses a random number function:

```
var random = function(seed) { /* ... */ }
let m = macro {
    rule {()} => {
        var n = random(42);
        // ...
    }
}
```

This macro needs to refer to `random` in any context that it gets invoked. But its context could have a different binding to `random`!

```
function foo() {
    var random = 42;
    m ()
}
```

Hygiene needs to keep the two `random` bindings different. So sweet.js will expand this into something like:

```
var random$1 = function(seed) { /* ... */ }
function foo() {
    var random$2 = 42;
    var n = random$1(42);
    // ...
}
```

Note that there is no way for hygiene to do this if it only renamed identifiers inside of macros since both `random` bindings were declared outside of the macro. Hygiene is necessarily a whole program transformation.

By default sweet.js will rename every variable to something like `name$102`. This gives correct but somewhat messy generated code. Usually this is not a problem since sweet.js provides source maps (with the `--sourcemap` flag) so you rarely need to inspect the generated source. If you would like to have sweet.js generate cleaner code you can use the `--readable-names` flag which will only rename variables when it is absolutely necessary. This flag is not yet turned on by default since it only supports ES5 code at the moment.

4 Case Macros

Sweet.js also provides a more powerful way to define macros: case macros. Case macros allow you to manipulate syntax using the full power of JavaScript. Case macros look like this:

```
macro <name> {
  case { <pattern> } => { <body> }
}
```

Case macros look similar to rule macros with a few important differences. First, the patterns also match the macro name instead of just the syntax that comes after it:

```
macro m {
  case { $name $x } => { ... }
}
m 42  // `$name` will be bound to the `m` token
      // in the macro body
```

Most of the time you won't need to match the name and can use the wildcard

_ pattern (which matches any token but doesn't bind it to a name) to ignore it:

```
macro m {
  case { _ $x } => { ... }
}
```

The other difference from rule macros is that the body of a macro contains a mixture of templates and normal JavaScript that can create and manipulate syntax. For example, here's the identity macro defined with a case macro:

```
macro id {
  case {_ $x } => {
    return #{ $x }
  }
}
```

Templates are now created with the `#{...}` form (which is technically shorthand for `syntax {...}`). The `#{...}` form creates an array of *syntax objects* using any pattern bindings that are in scope (i.e. were matched by the pattern).

Syntax objects are the representation of tokens that sweet.js uses to keep track of lexical context (which is used to maintain hygiene). They can be created with templates (`#{}`) but you can also create individual syntax objects using the lexical context of an existing syntax object:

```
macro m {
  case {_ $x } => {
    var y = makeValue(42, #{$x});
    return [y]
  }
}
m foo
// --> expands to
42
```

4.1 Creating Syntax Objects

Sweet.js provides the following functions to create syntax objects:

- `makeValue(val, stx)` -- `val` can be a `boolean`, `number`, `string`, or `null`/`undefined`
- `makeRegex(pattern, flags, stx)` -- `pattern` is the string representation of the regex pattern and `flags` is the string representation of the regex flags
- `makeIdent(val, stx)` -- `val` is a string representing an identifier
- `makePunc(val, stx)` -- `val` is a string representing a punctuation (e.g. `=`, `,`, `>`, etc.)
- `makeDelim(val, inner, stx)` -- `val` represents which delimiter to make and can be either `"()"`, `"[]"`, or `"{}"` and `inner` is an array of syntax objects for all of the tokens inside the delimiter.

If you want strip a syntax object of its lexical context and get directly at the token you can use `unwrapSyntax(stx)`.

Case macros also provide `throwSyntaxError(name, message, stx)` when you need to throw an error inside of a case macro. `name` is a string that lets you name the error, `message` is a string to describe what went wrong, and `stx` is a syntax object or array of syntax objects that is used to print out the line number and surrounding tokens in the error message.

4.2 letstx

When using syntax object creation functions, it is convenient to refer to them in `#{}` templates. To do this sweet.js provides `letstx`, which binds syntax objects to pattern variables:

```
macro m {
  case {_ $x } => {
    var y = makeValue(42, #{$x});
```

```
    letstx $y = [y], $z = [makeValue(2, #{$x})];
    return #{$x + $y - $z}
  }
}
m 1
// --> expands to
1 + 42 - 2
```

The left-hand side of the `=` in a `letstx` is actually a pattern so you can, for example, match a repeating pattern:

```
macro m {
  case {_} => {
    letstx $x ... = [makeValue(1, #{here}),
                     makeValue(2, #{here}),
                     makeValue(3, #{here})];
    return #{
      [$x (,) ...]
    }
  }
}
m
// expands to:
// [1, 2, 3]
```

5 Extending Pattern Classes

Consider the following macro that matches against color options:

```
macro color_options {
  rule { (red) } => { ["#FF0000"] }
  rule { (green) } => { ["#00FF00"] }
  rule { (blue) } => { ["#0000FF"] }
}
r = color_options (red)
g = color_options (green)
// expands to:
// r = ["#FF0000"]
// g = ["#00FF00"]
```

While this macro seems to work, attempting to generalize it quickly leads to a

mess:

```
macro color_options {
    rule { (red, red) } => {
        ["#FF0000", "#FF0000"]
    }
    rule { (red, green) } => {
        ["#FF0000", "#00FF00"]
    }
    rule { (red, blue) } => {
        ["#FF0000", "#0000FF"]
    }
    // ... etc.
}
r = colors_options (red, green)
g = colors_options (green, blue)
// expands to:
// r = ["#FF0000", "#00FF00"]
// g = ["#00FF00", "#0000FF"]
```

While it is possible to solve this problem through the use of case macros, the declarative intent is quickly lost in a mess of token manipulation code.

5.1 The Invoke Pattern Class

Sweet.js provides a solution to this problem with the `:invoke` pattern class. This pattern class takes as a parameter a macro name which is inserted into the token tree stream before matching. If the inserted macro successfully matches its arguments, the result of its expansion is bound to the pattern variable. This makes declarative options simple to write:

```
macro color {
    rule { red } => { "#FF0000" }
    rule { green } => { "#00FF00" }
    rule { blue } => { "#0000FF" }
}
macro colors_options {
    rule { ($opt:invoke(color) (,) ...) } => {
        [$opt (,) ...]
    }
```

```
}
colors_options (red, green, blue, blue)
// expands to:
// ["#FF0000", "#00FF00", "#0000FF", "#0000FF"]
```

Invoke will only expand the parameterized macro once; if the macro expands to another macro it will not be expanded before being bound to the pattern variable.

If you need to keep expanding, use `:invokeRec` which will continue to expand as long as a macro name is returned as the first token. This is useful if you have a recursive macro but is not the default because it can be unintuitive if the macro is not a recursive macro. For example,

```
macro a {
  rule {} => { 1 + 2 }
}
macro b {
  rule { $x } => { a + $x }
}
macro c {
  rule { $y:invokeRec(b) } => { [$y] }
}

c 3
// expands to:
// [1 + 2] + 3
```

The `b` macro returns `a + $x` as its result. Since `a` is another macro which returns `1 + 2` as a result, so `1 + 2` is what gets loaded into the `$y` pattern variable, and the `+ 3` is surprisingly pushed outside the scope of the macro.

This is a change from the default behavior of version 0.5.0 in which `:invoke` behaved like `:invokeRec` and `:invokeOnce` was available for the current default behavior.

5.2 Implicit Invoke

Custom pattern classes act as an implicit invoke. This means that `$opt:invoke(color)` is equivalent to `$opt:color`:

```
macro color {
    rule { red } => { "#FF0000" }
    rule { green } => { "#00FF00" }
    rule { blue } => { "#0000FF" }
}
macro colors_options {
    rule { ($opt:color (,) ...) } => {
        [$opt (,) ...]
    }
}
colors_options (red, green, blue, blue)
// expands to:
// ["#FF0000", "#00FF00", "#0000FF", "#0000FF"]
```

Custom pattern classes only support single-token identifiers. If you need to call a multi-token name or a punctuator name, you should explicitly call `$foo:invoke(...)` with the name.

5.3 Identity Rules

Sometimes you want your custom operator to just return exactly what it matched rather than transform it into something else. Identity rules let you do just that. If you leave off the body of a rule, it will just return exactly the syntax that the rule matched:

```
macro func {
  rule { function ($args (,) ...) { $body ... } }
  rule { function $name:ident ($args (,) ...) { $body ... } }
}

macro checkFunc {
  rule { $f:func } => { $f }
}

x = checkFunc function() {}
// expands to:
```

```
// x = function() {}
```

5.4 Errors for Invoke

The function `throwSyntaxCaseError` is similar to `throwSyntaxError`, but should be used specifically when you are using `:invoke`. Internally, sweet.js throws a SyntaxCaseError when a macro fails to match so `throwSyntaxCaseError` just lets you do it manually. Here's how you might use it in a macro that checks for keyword:

```
macro keyword {
  case { _ $kw } => {
    var kw = #{ $kw };
    if (kw[0].token.type === parser.Token.Keyword) {
      return kw;
    }
    throwSyntaxCaseError('Not a keyword');
  }
}
```

6 Let macros

Sometimes you don't want a macro to recursively call itself. For example, say you want to override `function` to add some logging information before the rest of the function executes:

```
macro function {
  case {_ $name ($params ...) { $body ...} } => {
    return #{
      function $name ($params ...) {
        console.log("Imma let you finish...");
        $body ...
      }
    }
  }
}
```

If you tried to run this through the compiler it will loop forever since the `function` identifier in the macro expansion is bound to the `function` macro. To prevent this you can use the `let` macro binding form:

```
let function = macro {
  case {_ $name ($params ...) { $body ...} } => {
    return #{
      function $name ($params ...) {
        console.log("Imma let you finish...");
        $body ...
      }
    }
  }
}
```

This binds `function` to the macro in the rest of the code but not in the body of the `function` macro.

7 Infix Macros

Sweet.js also lets you match on previous syntax using `infix` rules. Use a vertical bar (`|`) to separate your left-hand-side from your right-hand-side.

```
macro unless {
  rule infix { return $value:expr | $guard:expr } => {
    if (!($guard)) {
      return $value;
    }
  }
}

function foo(x) {
  return true unless x > 42;
  return false;
}
```

You can use the `infix` keyword with procedural macros, too. The macro name is just the first token on the right-hand-side.

```
macro m {
  case infix { $lhs | $name $rhs } => { ... }
}
```

Infix rules can be mixed and matched with normal rules:

```
macro m {
  rule infix { $lhs | $rhs } => { ... }
  rule { $rhs } => { ... }
}
```

You can even leave off the right-hand-side for a postfix macro:

```
macro m {
  rule infix { $lhs | } => { ... }
}
```

Sweet.js does its best to keep you from clobbering previous syntax.

```
macro m {
  rule infix { ($args ...) | $call:expr } => {
    $call($args ...)
  }
}

(42) m foo; // This works
bar(42) m foo; // This is a match error
```

The second example fails to match because you'd be splitting the good function call on the left-hand-side in half. The result would have been nonsense, giving you a nasty parse error.

8 Custom Operators

Custom operators let you define your own operators or override the built-in operators. They are similar to infix macros except rather than matching arbitrary syntax before and after the identifier name, both the left and right

operands must be valid JavaScript expressions. You can think of them as infix macros with a pattern of `{ $left:expr | $right:expr }`. This limitation however means that you can define precedence and associativity for your custom operator.

There are two definition forms. One for binary operators and one for unary operators:

```
// binary operators
operator <name> <precedence> <associativity>
    { <left operand>, <right operand> } => #{ <template> }

// unary operators
operator <name> <precedence>
    { <operand> } => #{ <template> }
```

- `<name>` can be any valid macro name (ie. identifiers and punctuation but not delimiters, multi token names must be wrapped in parentheses)
- `<precedence>` is a number (higher numbers bind more tightly)
- `<associativity>` is either `left` or `right`
- `<left operand>`/`<right operand>`/`<operand>` is a pattern variable (eg `$left`) that will be bound in the template
- `<template>` is what the operator will expand into

For example, the following defines `^^` to be the power operator:

```
operator (^^) 14 right
    { $base, $exp } => #{ Math.pow($base, $exp) }

y + x ^^ 10 ^^ 100 - z
// expands to:
// y + Math.pow(x, Math.pow(10, 100)) - z;
```

The precedence of `^^` (14) is higher than the precedence of `+` and `-` (12, see the chart in the next section) so `^^` binds more tightly. Since we defined `^^` to be right associative, `x ^^ 10 ^^ 100` is equivalent to `x ^^ (10 ^^ 100)`. Left

associative would have meant `(x ^^ 10) ^^ 100`.

8.1 Examples

Custom operators and infix macros complement each other nicely.

```
macro (=>) {
    rule infix { $param:ident | $body:expr  } => {
        function ($param) { return $body }
    }
}
operator (|>) 1 left { $l, $r } => #{ $r($l) }

var res = 10 |> x => x * 2
             |> y => y - 3
// expands to:
// var res = function (x) {
//         return function (y) {
//             return y - 3;
//         }(x * 2);
//     }(10);
```

Or let's say we want to chain promises:

```
macro (=>) {
    rule infix { ($params ...) | { $body ... } } => {
        function ($params ...) { $body ... }
    }
}
operator (>>=) 12 left { $l, $r } => #{$l.then($r) }

getPromise('test.json')  >>= JSON.parse >>= (response) => {
  console.log("JSON Response!", response);
}
// expands to:
// getPromise('test.json').then(JSON.parse).then(function (response) {
//     console.log('JSON Response!', response);
// });
```

Custom operators also let you change the behavior of the built-in operators:

```
operator + 12 left { $l, $r } => #{ add($l, $r) }
function add(x, y) {
```

```
    // custom addition
}

100 + x - y * 5 + 30
// expands to
// function add(x, y) {
//     // custom addition
// }
// add(add(100, x) - y * 5, 30);
```

You can even fix ==!

```
operator == 9 left { $l, $r } => #{ $l === $r }

if ("42" == 42) {
    // never runs!
}
// expands to:
// if ('42' === 42) {
//    // never runs!
// }
```

Keep in mind that redefining the built-in operators needs to be done with care. While it's tempting to fix some of the *wat* implicit conversions of ==/+/- and company, this could lead to hard to understand code. It should be used sparingly if at all. With great power...

8.2 Operator Precedence

The following charts note the precedence and associativity of the built-in operators. A higher precedence number means the operator binds more tightly.

8.2.1 Unary Operators

Operator	Precedence
`new`	16

Operator	Precedence
++	15
--	15
!	14
~	14
+	14
-	14
typeof	14
void	14
delete	14
yield	2

8.2.2 Binary Operators

Operator	Precedence	Associativity
*	13	left
/	13	left
%	13	left
+	12	left
-	12	left
>>	11	left
<<	11	left
>>>	11	left
<	10	left
<=	10	left
>	10	left
>=	10	left
in	10	left
instanceof	10	left
==	9	left
!=	9	left

Operator	Precedence	Associativity
`!==`	9	left
`&`	8	left
`^`	7	left
`|`	6	left
`&&`	5	left
`||`	4	left

9 Reader Extensions

The reader is what turns a string of code into tokens for the expander to work with. While it doesn't know anything about the semantics of JavaScript, it parses the code into tokens with several assumptions tailored to JavaScript. For example, it reads `/abc/` as a single RegExp token, but it reads `abc/d` as three distinct tokens: an identifier, a punctuator, and another identifier. In fact, the algorithm for determining when `/` starts a regex or is a simple division without fully parsing the code is a key breakthrough that enables macros in JavaScript.

The default reader separates code into identifiers, literals, and punctuators. It also matches delimiters (`{}()[]`) and creates a *tree* of tokens, so everything inside matching delimiters exist as children of a delimiter token.

You may want to extend the behavior of the reader, just like you would extend the semantics of a language with macros. Doing so allows you to embed custom syntax that may not even be valid according to the default reader, for example changing how the forward slash `/` is handled. For the most part, you will use macros to extend JavaScript, but it might be useful to control exactly how the reader parses tokens.

sweet.js allows reader customization through **readtables**. A readtable is simple mapping from single characters to procedures that do the reading when the specified character is encountered in the source. For example, you can install a function to be called when `<` is encountered, and you are responsible for reading as much of the source as you need and producing tokens.

Readtables allow composable extensions to the tokenizing phase, similar to how macros allow composable extensions to the parsing phase. You create a new readtable by extending an existing one. For example, here's how you extend the existing readtable with new functionality for `<`:

```
var readtable = sweet.currentReadtable().extend({
    '<': function(ch, reader) {
        // read stuff here
    }
});
```

And you can set the readtable programmatically:

```
sweet.setReadtable(readtable);
```

Or load it via a command line option. If you do this, the exported value of the module must be the readtable:

```
sjs --load-readtable ./readtable.js

# Or the shorthand
sjs -l ./readtable.js
```

By continually extending the current readtable, we can install the extensions in a modular way and build up a final readtable with our desired behaviors. You can pass multiple `-l` (or `--load-readtable`) flags to load multiple readtables extensions. If multiple readtables with the same character are installed, the last one loaded wins. There is currently no way to handle these kinds of

collisions; every character must have a single unique handler (or none).

9.1 Readtables

A readtable is an object that maps characters to handler functions, and an `extend` method. Internally, a base readtable exists which is the current readtable by default. To create a new one, you always extend an existing one as explained above:

```
var readtable = sweet.currentReadtable().extend({
    '<': function(ch, reader) {
        reader.readPunctuator();
        var token = reader.readToken();
        if(token.type !== reader.Token.Identifier) {
          return null;
        }

        // Continue parsing and return a single token or
        // an array of tokens
    }
});
```

Handler functions take two arguments: the character that invoked the handler and a reader object. You access the reader API through this reader object.

Currently, readtables are only invoked on punctuators. That means you can't customize what happens on a double quote `"` or a delimiter like `(`. This might be possible in the future.

The reader has not read the punctuator that invoked your handler, so you must always read past it with `reader.readPunctuator()` first.

You can return `null` from a handler function and the reader will automatically reset the state of the parser and continue reading normally. This lets you parse grammar that is ambiguous, and when something doesn't match you can

simply bail. If you want to throw an error instead, you can use `throwSyntaxError` in the reader API to show a nice error message to the user.

Lastly, you can return either a single token or an array of tokens from the handler. If you return multiple tokens, `readToken` will only return the first one and queue up the rest.

There are two functions on the sweet module for working with readtables:

- `sweet.currentReadtable()` — Gets the current readtable, which always has an `extend` method on it to install new extensions.

- `sweet.setReadtable(rt)` — Mutates the reader state to use the specified readtable when reading. Most likely users will use one or more `-l` or `--load-readtable` flags to `sjs`, but you can set it progammatically with this method.

9.2 Reader API

A reader object is always passed as the second argument when invoking a reader extension. It has several properties that are readable and writable:

- `reader.source` — The full raw string of the code being read
- `reader.length` — The length of `reader.source`
- `reader.index` — The current position of the reader into the source
- `reader.lineNumber` — The current line number
- `reader.lineStart` — The index into the source which the current line began
- `reader.extra` — read-only; an object that holds temporary data like comments

The following functions are also available on the reader object. You can read tokens and skip comments with these. All of this functions (except `skipComment`) with throw an error if it fails to read a valid token. You can use `suppressReadError` to suppress this (explained more below).

- `reader.readIdentifier()` — Read an identifier (variable name, keyword, null and boolean literals)
- `reader.readPunctuator()` — Read a punctuator
- `reader.readStringLiteral()` — Read a string literal
- `reader.readNumericLiteral()` — Read a numeric literal
- `reader.readRegExp()` — Read a regular expression
- `reader.readDelimiter()` — Read a delimiter, including everything inside it and up to the matching end delimiter
- `reader.readToken()` — Read a token of any type. Note that this can trigger another readtable invocation if the current punctuator has an entry in the current readtable. If you use this, your reader extension could potentially be called recursively. In addition, you might need to be aware of how readtables queue tokens (see `peekQueued` below).
- `reader.skipComment()` — Skip past a comment and any whitespace, if it exits.

The following functions are available for manually constructing tokens. You give it the parsed value and any options. The constructed tokens will automatically be given source location information, but it can't automatically infer the range. You should pass where the token theoretically started in the original source with the `start` option. So a common pattern looks like this:

```
var start = reader.index;
// do some reading
return reader.makeIdentifier('foo', { start: start });
```

`makeDelimiter` does not take any options because it can infer the range from

the first and last inner tokens.

- `reader.makeIdentifier(value, opts)` — Make an identifier token
- `reader.makePunctuator(value, opts)` — Make a punctuator token
- `reader.makeStringLiteral(value, opts)` — Make a string literal token. It takes an addition boolean parameter `octal` in `opts`
- `reader.makeNumericLiteral(value, opts)` — Make a numeric literal token
- `reader.makeRegExp(value, opts)` — Make a regexp token
- `reader.makeDelimiter(value, inner)` — Make a delimiter token with the token array `inner` as its children

The following functions are simple utility functions:

- `reader.isIdentifierStart(charCode)` — Check if `charCode` is a valid character for starting an identifier
- `reader.isIdentifierPart(charCode)` — Check if `charCode` is a valid non-starting character for an identifier
- `reader.isLineTerminator(charCode)` — Check if `charCode` is a character that terminates a line
- `reader.peekQueued(offset)` — Inspects the tokens on the reader queue, returning the nth (based on `offset`) token from the front (0 is the next token). Reader extensions are invoked from `readToken`, but can return multiple tokens, so the rest are pushed onto a queue. You may need to be aware of this with recursive reader extensions or other complicated scenarios.
- `reader.getQueued()` — Gets the next token from the reader queue and removes it from the queue
- `reader.suppressReadError(readFunc)` — Suppresses exceptions from any of the `read*` functions and returns `null` instead if something went wrong. Example: `reader.suppressReadError(reader.readIdentifier)`
- `reader.throwSyntaxError(name, message, token)` — Throws a syntax error.

`name` is just a general identifier for your project, and `message` is the specific message to display. You must pass a `token` so it can properly locate where the error occurred in the original source.

If you want to look at some examples, check out the readtables tests.

10 Modules

10.1 Using Modules

At the moment sweet.js supports a primitive form of module support with the `--module` flag.

For example, if you have a file `macros.js` that defines the `m` macro:

```
// macros.js

macro m { /* ... */ }
export m;
```

and `my_sweet_code.js` uses `m`:

```
// my_sweet_code.js

m 42
```

You would compile this with:

```
$ sjs --module ./macros.js  my_sweet_code.js
```

Note that modules must use the `export` keyword. This allows modules to define "private" macros that are not visible to the main code.

The `--module` flag uses the node path to look up the module file so you can

publish and use macro modules on npm. Checkout lambda-chop for an example of this.

The biggest limitation with the current approach is that you can't arbitrarily interleave importing compile-time values (macros) and run-time values (functions). This will eventually be handled with support for "proper" modules (issue #43).

10.2 Node Loader

If you'd like to skip using the `sjs` binary to compile your sweet.js code, you can use the node loader. This allows you to `require` sweet.js files that have the `.sjs` extension:

```
var sjs = require('sweet.js'),
    example = require('./example.sjs');

example.one;
```

Where ./example.sjs contains:

```
//  example.sjs
macro id {
    rule { ($x) } => {
        $x
    }
}

exports.one = id (1);
```

Note that `require('sweet.js')` must come before any requires of `.sjs` code. Also note that this does not import any macros, it just uses sweet.js to compile files that contain macros before requiring them.

Alternatively, you can use `sweet.loadMacro` to achieve a similar effect to the

`--module` command line flag:

```
var sweet = require('sweet.js');
// load all exported macros in `macros/str.sjs`
sweet.loadMacro('./macros/str');
// test.sjs uses macros that have been defined and exported in `macros/str.sjs`
require('./test.sjs');
```

This is basically equivalent to running `sjs --module ./macros/str test.sjs`.

11 Case macro API

Functions available inside of a case macro.

11.1 `makeValue(val, stx)`

Returns a syntax object with the lexical context of `stx` and the value of `val`. `val` can be a `boolean`, `number`, `string`, or `null`/`undefined`.

11.2 `makeRegex(pattern, flags, stx)`

Returns a syntax object with the lexical context of `stx` and the regular expression literal pattern of `pattern` and flags of `flags`. `pattern` is a string representation of the regex pattern and `flags` is the string representation of the regex flags.

11.3 `makeIdent(val, stx)`

Returns a syntax object with the lexical context of `stx` and the identifier name of `val`. `val` is a string representing an identifier.

11.4 makePunc(val, stx)

Returns a syntax object with the lexical context of `stx` and the punctuator value of `val`. `val` is a string representing a punctuation token (e.g. `=`, `,`, `>`, etc.).

11.5 makeDelim(val, inner, stx)

Returns a syntax object with the lexical context of `stx` and delimiter type of `val` and the inner syntax objects of `inner`. `val` represents which delimiter to make and can be either `"()"`, `"[]"`, or `"{}"` and `inner` is an array of syntax objects for all of the tokens inside the delimiter.

11.6 unwrapSyntax(stx)

Returns the value of the given syntax object.

```
"foo" === unwrapSyntax(makeIdent("foo", null))
42    === unwrapSyntax(makeValue(42, null))
```

11.7 localExpand(stx)

Force the expansion of all macros in the array of syntax objects `stx`. Returns an array of syntax objects.

```
macro PI { rule {} => { 3.14159 }}

macro ex {
  case { { $toks ... } } => {
    var expanded = localExpand(#{$toks ...});
    assert(unwrapSyntax(expanded[0]) === 3.14159)
    return expanded;
  }
}

ex { PI }
```

Note that `localExpand` is currently experimental. If you're coming from Racket you might also be expecting a stop list which is not yet implemented.

12 Compiler API

12.1 `sweet.compile`

Expands all macros and return the expanded code as a string.

```
(Str, {
        sourceMap: Bool,
        filename: Str,
        readableNames: Bool,
        maxExpands: Num
    }) -> {
        code: Str,
        sourceMap: Str
    }
sweet.compile(code, options)
```

Parameters:

- `code` the code to expand
- `options` options object:
  - `sourceMap` if `true` generate a source map
  - `filename` file name of the source file to go into the source map
  - `readableNames` clean up variables that were renamed from hygiene (`foo$100` becomes `foo` where ever possible). Only supports ES5 code.
  - `maxExpands` maximum number of times to expand macros. Used to implement macro stepping.

Return:

An object with two fields:

- `code` the expanded code
- `sourceMap` the source map

12.2 `sweet.parse`

Expands all macros and returns an AST.

```
(Str, [...[...Syntax]], { maxExpands: Num }) -> AST
sweet.parse(code, modules, options)
```

Parameters:

- `code` the code to expand
- `modules` each array should be the result of `expand` on a module. Any macros in the module that were `export`ed will be in scope while expanding `code`.
- `options` options object:
  - `maxExpands` maximum number of times to expand macros. Used to implement macro stepping.

Return:

The abstract syntax tree. See the Parser API for details.

12.3 `sweet.expand`

Expands all macros and returns an array of syntax objects.

```
(Str, [...[...Syntax]], { maxExpands: Num }) -> [...Syntax]
sweet.expand(code, modules, options)
```

Parameters:

- `code` the code to expand
- `modules` each array should be the result of `expand` on a module. Any macros in the module that were `export`ed will be in scope while expanding `code`.
- `options` options object:
  - `maxExpands` maximum number of times to expand macros. Used to implement macro stepping.

Return:

An array of syntax objects. Syntax objects are an object with two fields:

- `token` which is a token object that esprima understands.
- `context` holds hygiene information.

13 FAQ

13.1 How do I Run Sweet.js in the Browser?

Load sweet.js using AMD/require.js:

```
require(["./sweet"], function(sweet) {
    // ...
});
```

Then just use the sweet.js compiler API. You can see an example of this in action with the sweet.js editor here and here.

13.2 How do I break hygiene?

Sometimes you really do need to break the wonderful protections provided by hygiene. Breaking hygiene is usually a bad idea but sweet.js won't judge.

Breaking hygiene is done by stealing the lexical context from syntax objects in the "right place". To clarify, consider `aif` the anaphoric if macro that binds its condition to the identifier `it` in the body.

```
var it = "foo";
long.obj.path = [1, 2, 3];
aif (long.obj.path) {
  console.log(it);
}
// logs: [1, 2, 3]
```

This is a violation of hygiene because normally `it` should be bound to the surrounding environment (`"foo"` in the example above) but `aif` wants to capture `it`. To do this we can create an `it` binding in the macro that has the lexical context associated with the surrounding environment. The lexical context we want is actually found on the `aif` macro name itself. So we just need to create a new `it` binding using the lexical context of `aif`:

```
macro aif {
  case {
    // bind the macro name to `$aif_name`
    $aif_name
    ($cond ...) {$body ...}
  } => {
    // make a new `it` identifier using the lexical context
    // from `$aif_name`
    var it = makeIdent("it", #{$aif_name});
    letstx $it = [it];
  return #{
      // create an IIFE that binds `$cond` to `$it`
      (function ($it) {
          if ($cond ...) {
          // all `it` identifiers in `$body` will now
          // be bound to `$it`
          $body ...
        }
      })($cond ...);
```

```
        }
    }
}
```

13.3 How do I output comments?

Comments in a rule macro or inside a `#{...}` template should "just work". If you want to create comment strings programmatically you can use a token's `leadingComments` property.

```
macro m {
    case {_ () } => {
        var x = makeValue(42, #{here});
        x.token.leadingComments = [{
            type: "Line",
            value: "hello, world"
        }];
        return withSyntax ($x = [x]) #{
            $x
        }
    }
}
m()
```

will expand to

```
//hello, world
42;
```

13.4 How do I convert a token to a string literal?

Often times you'll have an identifier or numeric literal token that you'd like to convert to a string literal. This little helper macro does just that:

```
macro to_str {
  case { _ ($toks ...) } => {
    return [makeValue(#{ $toks ... }.map(unwrapSyntax).join(''), #{ here })];
  }
}
```

```
to_str(1 foo "bar")
// expands to:
// '1foobar'
```

13.5 How do I debug macros?

13.5.1 Stepping Through Expansion

You can use `sjs --num-expands <number>` to walk through macro expansion one step at a time. This is particularly helpful when writing recursive macros.

The editor also supports stepping.

13.5.2 Understanding Case Macros

If you're trying to understand how a case macro is working two useful techniques are logging syntax objects to see what they actually contain and inserting `debugger` statements to pause the debugger during expansion.

```
macro m {
    case { _ $x } => {
        console.log(#{$x})
        debugger;
        return #{42}
    }
}
m 100
```