Adding Syntax Parameters to the Sweet.JS Macro Library for JavaScript

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Vimal Kumar

May 2015

The Designated Project Committee Approves the Project Titled

Adding Syntax Parameters to the Sweet.JS Macro Library for JavaScript

by

Vimal Kumar

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2015

Dr. Thomas H. Austin	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Dr. Ronald Mak	Department of Computer Science

ABSTRACT

Adding Syntax Parameters to the Sweet.JS Macro Library for JavaScript

by Vimal Kumar

Lisp and Scheme have demonstrated the power of macros to enable programmers to evolve and craft languages. Mozilla Sweet.JS provides a way for developers to enrich their JavaScript code by adding new syntax to the language through the use of macros. Sweet.JS provides the possibility to define hygienic macros inspired by Scheme. In this paper I present the implementation of a "syntax parameter" feature to Sweet.JS library. Syntax parameter is a mechanism for rebinding a macro definition within the dynamic context of a macro expansion. In my implementation I define the "syntaxparam," which defines and binds the syntax parameter part of the compiler; "syntaxLocalValue," which pull the syntax parameter definition in the defined scope and "replaceSyntaxParamTransform," which expand the syntax parameter macro definition defined within the macro body using "syntaxLocalValue".

ACKNOWLEDGMENTS

This project would not have been possible without the kind support and help of many individuals. I would like to extend my sincere thanks to all of them.

I am highly indebted to Dr. Thomas Austin,and Tim Disney, for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project. I would like to thank my thesis committee members Dr. Chris Pollett and Dr. Ronald Mak for their encouragement, insightful comments, and hard questions.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 What are macros?

A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence according to some defined procedure. Using a macro system a programmer can introduce new syntactic elements to the programming language.Macros found in a program are expanded by a *macro expander* and allow a programmer to enable code reuse. There are two types of macro systems

1. **Lexical macro systems**, such as the C preprocessor,transform the codes before compilation. Lexical macros are ignorant of the grammar of the core programming language and therefore sometimes result in ill-formed programs and in accidental capture of identifier. They only require lexical analysis, that is they operate on the source text prior to nay parsing, by simple substitution of tokenized character sequence for other tokenized character sequences, according to user-defined rules. Consider this example Example

```
#define INCI(i) {int a=0; ++i;}
int main(void)
        {
        int a = 0, b = 0;
        INCI(a);
        INCI(b);
        printf("a is now %d, b is now %d\n", a, b);
        return 0;
```

```
        }
```

Running through C preprocessor result in

```
int main(void)
        {
        int a = 0, b = 0;
        {int a=0; ++a;};
        {int a=0; ++b;};
        printf("a is now %d, b is now %d\n", a, b);
        return 0;
        }
```

The variable a declared in the top scope is shadowed by the a variable in the macro, which introduce a new scope. Output of the compiled program

```
a is now 0, b is now 1
```

2. **Syntatic macro systems**,like these in the Lisp and Scheme programming languages are aware of the grammar of the core programming language. They transform the syntax tree according to a number of user-defined rules. Rules can be written in the same programming language as the program or another language that relies on a fully external language to define the transformation, such as the XSLT preprocessor for XML. Below is an example of a syntatic macro in scheme that swaps the values.

```
(define-syntax-rule (swap x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
```

define-syntax-rule is the template, used in place of a form that matches the pattern, except that each instance of a pattern variable in the template is replaced with the part of the macro use the pattern variable matched. For example,

```
(let ([tmp 5]
      [other 6])
  (swap tmp other)
  (list tmp other))
```

The result of the above expression should be (6 5). The naive expansion of this use of swap, however, is

```
(let ([tmp 5]
      [other 6])
  (let ([tmp tmp])
    (set! tmp other)
    (set! other tmp))
  (list tmp other))
```

whose result is (5 6). The problem is that the naive expansion confuses the *tmp* in the context where swap is used with the *tmp* that is in the macro template. Instead it produces

```
(let ([tmp 5]
      [other 6])
  (let ([tmp_1 tmp])
    (set! tmp other)
    (set! other tmp_1))
  (list tmp other))
```

with the correct result in (6 5). RacketâĂŹs pattern-based macros automatically maintain lexical scope,so macro implementors can reason about variable reference in macros and macro uses in the same way as for functions and function calls.

1.2 What Sweet.JS bring for us?

Sweet.JS is a hygienic macro compiler for JavaScript that takes JavaScript macros and produces normal JavaScript code which one can run in a browser or using a standalone interpreter like Node.JS. The idea is that you define a macro with a name and a list of patterns. Whenever a macro is invoked, the code is matched and expanded at the compile time.

Sweet.JS provides the two ways to define a macro: simple pattern based *rule* macros that work by matching a syntax pattern and generating the new pattern based on the template and the more powerful procedural *case* macros allow you to manipulate syntax.Below example shows the rule based macro

```
macro define {
 rule { $x } => {
            var $x
 }
 rule { $x = $expr } => {
      var $x = $expr
}
}
define y;
define y = 5;
```

Above code will expand to

```
var y;
var y = 5;
```

1.3 What is macro hygiene?

Hygienic macros are macros whose expansion does not to cause the accidental capture of identifier introduced by the macro expander. Hygiene prevents variable names inside the macros from clashing with the variables in the surrounding code. Hygiene macro systems are the feature of programming language such as Scheme and Dylan.

There are occasions when traditional hygienic binding is insufficient: one one example is the "anaphoric if condition", where while expanding the macro definition at compile time may introduce new variable bindings,that capture variables in your

own code. That is, the new binding might shadow a variable that you have already created. Example

```
(define-syntax or
(syntax-rules ()
((_ e1 e2)
(let ((t e1))
(if t t e2)))))
```

Calling above macro

```
(let ((t 5))
        (or #f t))
```

Let expand the above macro

```
(let ((t 5))
        (let ((t #f))
(if t t t)))
```

This program evaluates to #f, which is not the desired output. On expanding the macro the binding "t" is shadowed to #f.If you run this in the scheme REPL , output will be 5. One way to work around this, which was a common trick for LISP programmers of yore is to choose variable names that a programmer is unlikely to guess. We could modify the macro expander to automatically rename any variables bound by a macro expansion. In this case, our simple test program would expand as follows, using our first definition of "or"

```
(let ((t 5))
 (let ((t.1 #f))
   (if t.1 t.1 t)))
```

This program evaluates as expected.

1.4 syntax-parameters in Racket?

Syntax-parameter are a mechanism for rebinding a macro definition with in the dynamic extent of a macro expansion.The ability to write functions that instead of accepting and returning values, accept and return pieces of source code, allows for abstractions and extensions that just simply aren't possible in other languages. Below example shows how to define the syntax parameter

```
#lang racket
(require racket/stxparam)
(define-syntax-parameter example-stx-parameter
    (lambda (stx)
        #'(displayln "I'm a syntax parameter!")))
```

All macros are functions that take as input a syntax object representing the piece of the program where it was located, and return a new syntax object to replace the old one with in the program. So with the following syntax parameter defined, this code

```
(example-stx-parameter)
```

The expanded code is

```
(displayln "I'm a syntax parameter!")
```

Remember that this happens at compile time and that this code re-write occurs before the code is run.The purpose of a syntax parameter is to be modified by other macros with the syntax-parameterize form, which looks like this

```
(syntax-parameterize
([example-stx-parameter (lambda (stx)
                          #'(displayln "I'm parameterized!"))])
    (example-stx-parameter))
(example-stx-parameter)
```

The syntax-parameterize form "renames" the parameter, giving it a new value that applies only in the code inside the syntax-parametrize form. So when the above code is expanded, it produces

```
(displayln "I'm parameterized!")
(displayln "I'm a syntax parameter!")
```

The occurence of “example-stx-parameter” inside the *syntax-parameterize* form used the function defined in syntax-parameterize to transform the code instead of the original function.

Here in this paper,I present the example of macro that break Sweet.JS hygienic macro and propose a solution taking inspiration from Scheme's syntax parameter. Problem and solution discussed in up-coming chapters.

CHAPTER 2

Basics of Sweet.JS

Sweet.js implements macros for JavaScript, which takes source code written with Sweet.JS macros and produces the expanded source that can be run in any JavaScript environment.

2.1 Type of macros in Sweet.JS

1. **Rule macros** Rule macros work by matching a syntax pattern and generating new syntax based on the template. To define a rule base macro the grammar is

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

The following macro defines swapping of two number

```
macro swap {
rule { ($x, $y) } => {
        var tmp = $x;
        $x = $y;
        $y = tmp;
}
}
var foo = 5;
var tmp = 6;
swap(foo, tmp);
```

When the compiler hits "swap", it invokes the macro and runs each rule against the code after it. When a pattern is matched, it returns the code within the rule. You can bind identifiers & expressions within the matching pattern and use them within the code.

if Sweet.js did not support hygiene, this macro might expand to

```
var foo = 5;
var tmp = 6;
var tmp = foo;
foo = tmp;
tmp = tmp;
```

The `tmp` created from the macro collides with my local `tmp`. This is a serious problem, but macros solve this by implementing hygiene. Basically they track the scope of variables during expansion and rename them to maintain the correct scope. Sweet.js fully implements hygiene so it never generates the code you see above. It would actually generate the following code

```
var foo = 5;
var tmp$1 = 6;
var tmp$2 = foo;
foo = tmp$1;
tmp$1 = tmp$2;
```

Notice how two different "tmp" variables are created. This makes it extremely powerful to create complex macros elegantly.

2. **Case macros** Case macro are analogous to syntax-case in Scheme. Case macro allow the macro author to use JavaScript code to procedurally create

and manipulate the syntax. To define case macro, the grammar is

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

Example

```
macro rand {
  case { _ $x } => {
    var r = Math.random();
   letstx $r = [makeValue(r)];
 return #{ var $x = $r }
        }
}


rand x;
```

The above code expand to

```
var x$123 = 0.8367501533161177;
```

The body of a macro contains a mixture of templates and normal JavaScript that can create and manipulate syntax The code within the “case” is run at expand-time and you use #{} to create "templates" that construct code just like the syntax in the rule macros.

2.2 How Sweet.JS work?

The JavaScript macro system, Sweet.JS includes a separate reader that converts a sequence of tokens into a sequence of token trees, Analogous to s-expressions in scheme,without feedback from the parser.

Figure 1: Sweet.JS anatomy.

The parser gives structure to unstructured source code. The lexer which converts a character stream to a token stream and a parser converts the token stream into an AST according to a context free grammar [2].The macro expander must sit in between the lexer and the parser.Here the reader records sufficiently history information in the form of token trees in order to decide how to parse the token, which is required to decide if token is a divisor or a regular expression. In traditional JavaScript compilers, the parser and lexer are intertwined, rather than run the entire program through the lexer once to get a sequence of tokens, the parser calls out to the lexer from a given grammatical context with a flag to indicate if the lexer should accept a regular expression or a divide operator and the input character are tokenized accordingly [2].

Example

```
macro id {
        case {_ $x } => {
                return #{ $x }
        }
}
id 42
```

```
[
        { type:3,value:"macro"},
        {type:3,value:"id"},
        {type:11,value:{},inner:
                                [{type:4,value:"case"},

{type:11,value:{},inner:[{type:3,value:"_"},{type:3,value:"$x"}]},
                                 {type:7,value:"=>"},

{type:11,value:{},inner:[{type:4,value:"return"},{type:7,value:"#"},

        {type:11,value:{},inner:[{type:3,value:"$x"}]}]
                                 }]
        }
]
```

Figure 2: Token tree.

Reader convert string of token to token tree,

```
[{
    type:"Program",
    body:[{type:"ExpressionStatement",expression:{type:"literal",value:42}
         }],
    error:[{..}]
 }]
```

Figure 3: Final AST from parser.

Parser give structure to unstructured source code, In parsers without macro systems this is usually accomplished by a lexer which convert a character stream to a token stream and a parser which convert the token stream into an AST according to a context-free grammar.

The approach used in Sweet.JS is *Enforestation*,first pioneered by Honu [4]. Enforestation means transformation,extracts the sequence of terms produced by the reader to create a term tree. Consider following let example

Enforestation begins by loading the let macro into the macro environment and converting the function declaration into a term tree.

```
macro let{
    rule { $id= $init:expr }=>{
        var $id=$init
        }
    }

    function foo(x){
        let y=40+2
        return x+y;
        }
    foo(100);
```

Figure 4: Macro Expansion.

```
<fn:foo,
params:(x),
body:{
     <var:x, init:<op:+,left:40,right:2 >
     <return: <op:+,left:x,right:y>
    }>
 <call:foo, args(100)>
```

Figure 5: Term tree.

A term tree is a kind of proto-AST that represent a partial parse of the program. As the expander passes through the token trees, it creates term trees that contain unexpanded sub trees that will be expanded once all macro definition have been discovered in the current scope.

2.3 Problem trying to solve?

Although the benefits of hygienic macros are well established, there are occasions when traditional hygienic bindings are insufficient. The classic example is "anaphoric conditionals" where the value of the tested expression is available as an IT bindings.When the condition is true, an IT identifier is automatically created and set to the value of the condition. An anaphoric macro is a type of programming macro that deliberately captures some form supplied to the macro which may be referred to by an anaphor (an expression referring to another).

```
macro aif {
    case {
        $aif_name
        ($cond ...) {$tru ...} else { $els ... }
    } => {
        letstx $it = [makeIdent("it", #{$aif_name})];
        return #{
            (function ($it) {
                if ($cond ...) {
                    $tru ...
                } else {
                    $els ...
                }
            })($cond ...);
        }
    }
}

macro unless {
    rule { ($cond ...) { $body ...} } => {
        while (true) {
            aif ($cond ...) {
                // `it` is correctly bound by `aif`
                console.log("loop finished at: " + it);
              // break;
            } else {
                $body ...
            }
        }
    }
}
unless (x) {
    // `it` is not bound!
    console.log(it)
}
```

Figure 6: Break hygienic in Sweet.JS.

In above example, I define "unless" macro which run until condition is met, which in turn call the anaphoric-if condition which introduces an anaphor IT which should bound to the result of the test clause.

```
1. while (true) {
2.    (function (it$2) {
3.        if (x) {
4.            // `it` is correctly bound by `aif`
5.            console.log('loop finished at: ' + it$2);
6.        } else {
7.            // `it` is not bound!
8.            console.log(it);
9.        }
10.    }(x));
11.    }
```

Figure 7: Macro expansion.

In above macro expansion [Figure 7], at line (8) identifier IT is not defined. Here we wish to introduce the identifiers deliberately breaking hygiene.

Same unhygienic macros are possible in Scheme as shown in the below example, mistake is if our new syntax introduces a variable that conflicts with one in the code surrounding our macro.

```
(define-syntax-rule (aif condition true-expr false-expr)
 (let ([it condition])
   (if it
       true-expr
       false-expr)))

 (aif #t (displayln it) (void))

it: undefined;
cannot reference an identifier before its definition
in module: 'program
```

When using syntax-parameterize, IT acts as an alias for TMP. The alias behavior is created by make-rename-transformer. **define-syntax-parameter**, binds the key-

word to the value obtained by evaluating the transformer. The transformer provides the default expansion for the syntax parameter. **syntax-parameterize**, adjust the keyword to use the values obtained by evaluating their transformer in the expansion of the expression.

```
(require racket/stxparam)
(define-syntax-parameter it
  (lambda (stx)
  (raise-syntax-error (syntax-e stx)
    "can only be used inside aif")))

(define-syntax-rule (aif condition true-expr false-expr)
  (let ([tmp condition])
  (if tmp
  (syntax-parameterize ([it (make-rename-transformer #'tmp)])
    true-expr)
    false-expr)))
```

"define-syntax-parameter," binds keyword to the value obtained by evaluating transformer. The transformer provide the default expansion for the syntax parameter."syntax-parameterize," adjust keyword to use the values obtained by evaluating their transformer in the expansion of the expression.

CHAPTER 3

Syntax parameters

Syntax parameter are a mechanism for rebinding a macro definition with the dynamic extent of a macro expansion. With syntax parameter instead of introducing the unhygienic binding each time we instead create binding for the keyword, which we can adjust later when we want the keyword to have different meaning. As no new binding is introduced so hygiene is preserved, This is similar to the dynamic binding mechanism that we have at run time, except that the dynamic binding only occurs during macro expansion.

3.1 Approach 1

In this approach I tried to use pre-processing approach similar to C, where I transform the code before the main compiler get hold of it. âĂIJSyntaxParameterâĂİ which replace and bind the parameter with mapped value in the particular defined macro scope. Here macro transformation happen during the parse phase by matching the pattern. To define syntax parameter in Sweet.js provide a new keyword that look something like this:

SyntaxParameter(<parameter>,<Mapped to>,<Scope/Macro Name>,<Macro definition>)

Example of usage is shown below

```
defineSyntaxParameter it { rule {} => { console.log(" to be used in aif") } }
macro aif {
    case {
        $aif_name
        ($cond ...) {$tru ...} else { $els ... }
    } => {
     SyntaxParameter(it, $cond ... , aif ,
        return #
         {
            (function () {
                 if ($cond ...) {
                    $tru ...
                } else {
                    $els ...
                }
            })

        })
    }
}

macro unless {
    case {
         $unless_name
         ($cond ...) { $body ...} } => {
 return #{
        while (true) {
            aif ($cond ...) {
                // `it` is correctly bound by `aif`
                console.log("loop finished at: " + it);

                            } else {
                $body ...
            }()
            if($cond ...) {break;}
        }}
    }
}
```

Figure 8: Approach 1.

```
x=2
unless (x) {
    // `it` is  bound! correctly
    console.log(it)
}
```

Figure 9: Calling unless macro

On expansion above macro looks like this

```
x = 2;
while (true) {
    (function() {
        if (x) {
            // `it` is correctly bound by `aif`
            console.log('loop finished at: ' + x);
        } else {
            // `it` is  bound! correctly
            console.log(x);
        }
    }());
    if (x) {
        break;
    }
}
```

Figure 10: it identifier is correctly bounded to $cond.. Of an anaphoric-if

Since, 'it' is correctly bounded to $cond.. as desired, but this approach has certain disadvantage, since this won't allow user to define macros with name "SyntaxParameter" consider like

```
macro SyntaxParameter {
  ...
}
```

So to fix this we first need to implement some primitive function that help us to create and manipulate the arbitrary compile time syntax transformation. Macros are compile time syntax transformation ,At the moment in Sweet.js the only way to create a syntax transformer is by defining a macro. A macro is really just a function that takes syntax and returns syntax (thus a syntax transformer). So when the "expander" encounters a macro definition it converts the body of the macro into a function and loads it into the compile time environment.

3.2 Approach 2

In this approach I, defined "syntaxparam" similar to define-syntax-parameter in Scheme, which load the primitive function in compile time environment,"syntaxLocalValue," which return the compile time value from the environment and "replaceSyntaxParam," which transform the identifier with the compile time value from the environment within the specified scope of the macro. Example shown below

```
syntaxparam it => (function(x) {
         if(x==null) {
            return "To be used in Anaphoric-If"
         }
           else {
               return x;
           }
         })


macro aif {
    case {$aif_name
        ($cond ...) {$tru ...} else { $els ... }
     } => {
        var stxId = syntaxLocalValue(#{it},#{$aif_name})
        var cond=stxId(#{$cond ...})

       replaceSyntaxParam("it",cond ,#{$aif_name})
       return #
         {
            (function () {
                 if ($cond ...) {
                    $tru ...
                } else {
                    $els ...
                }
            })

        }
    }
}
```

Figure 11: Approach 2.Syntax parameter implementation

```
macro unless {
    case {
         $unless_name
         ($cond ...) { $body ...} } => {
 return #{
        while (true) {
            aif ($cond ...) {
                // `it` is correctly bound by `aif`
                console.log("loop finished at: " + it);
                            } else {
                $body ...
            }
        }}
    }
}

x=2
unless (2+3) {
    // `it` is  bound!
    console.log(it)
}
```

Figure 12: Approach 2.Syntax parameter implementation contd..

```
x = 2;
while (true) {
    (function () {
        if (2 + 3) {
            // `it` is correctly bound by `aif`
            console.log('loop finished at: ' + 2 + 3);
        } else {
            // `it` is  bound!
            console.log(2 + 3);
        }
    });
}
```

Figure 13: Approach 2. Syntax parameter expansion

The main entry point into the expander is appropriately enough, expand function is primarily responsible for handling hygiene. The env param is a mapping from identifier to macro definitions and ctx is a mapping of names to names. The expand function delegates to expandToTermTree, responsible for converting the syntax to TermTrees and loading any macro definitions it finds into the new env map.

Its in the "expandToTermTree," it call enforest repeatedly until the entire token tree has been converted into a term tree. its in enforest function where "syntaxparam"

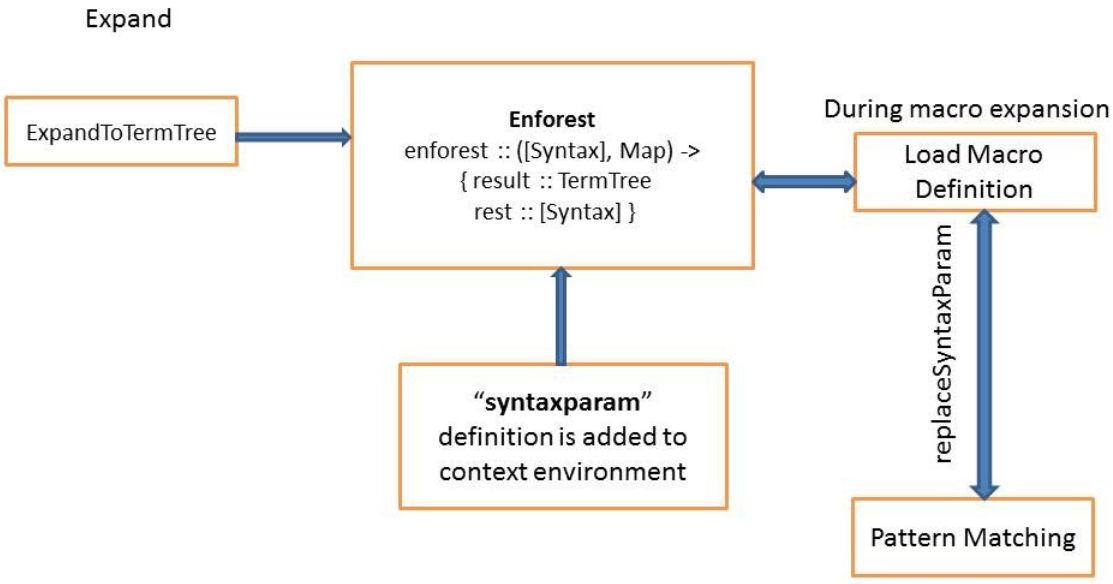

Figure 14: Approach 2. Implementation

definition loaded to context env map. When macro call is invoked it load the macro definition from the env in "loadmacrodef" function and during pattern matching "replaceSyntaxParam" which bind the identifier with the value.

LIST OF REFERENCES

[1] Eli, B., Culpepper, R., & Flatt, M. (2011, 11 05). Keep it Clean with Syntax Parameters,
http://scheme2011.ucombinator.org/papers/Barzilay2011.pdf

[2] Disney, T., Faubion, N., & Herman, D. (2014, 8 21). Sweeten Your JavaScript: Hygienic Macros for ES5,
http://disnetdev.com/papers/sweetjs.pdf

[3] Arai, H., & Wakita, K. (2012). An Implementation of A Hygienic Syntatic Macro System for JavaScript: A Preliminary Report,
http://www.is.titech.ac.jp/~wakita/files/arai-s3.pdf

[4] Rafkind, J., & Flatt, M. (2012, 9 26). Honu: Syntactic Extension for Algebraic Notation through Enforestation,
http://www.cs.utah.edu/plt/publications/gpce12-rf.pdf

[5] Flatt, M., Culpepper, R., & Findler, R. B. (2012, 3 27). Macros that Work Together,
http://www.cs.utah.edu/plt/publications/jfp12-draft-fcdf.pdf

[6] Disney, T. (n.d.). Retrieved from Sweet.JS Documentation,
http://sweetjs.org/doc/main/sweet.html