Spring 2015

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

Vimal Kumar
San Jose State University

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. 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. 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 for the Sweet.JS library. A syntax parameter is a mechanism for rebinding a macro definition within the dynamic context of a macro expansions thereby introducing implicit identifiers in a hygienic fashion. Some time hygienic macro bindings are insufficient such as with "anaphoric conditionals" where the value of the tested expression is available as an *it* binding. With syntax parameters, instead of introducing the binding unhygienically each time, we instead create one binding for the keyword, which we can then adjust later when we want the keyword to have a different meaning. As no new bindings are introduced hygiene is preserved.

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

1 Introduction 1

1.1 What are macros? 1

1.2 Hygiene 4

1.3 Syntax Parameters 6

2 Basics of Sweet.JS 9

2.1 Types of macros in Sweet.JS 9

2.2 Sweet.JS compilation process 13

2.3 Current limitation of Sweet.JS 15

3 Syntax parameters implementation for Sweet.JS 19

3.1 Pre-processing [Approach 1] 19

3.2 Syntax parameter processing at compile time [Approach 2] 22

APPENDIX

Code base 29

A.1 Syntax parameter processing at compile time [Code base] 29

A.2 Syntax parameter pre-processing [Code base] 34

LIST OF FIGURES

1	Sweet.JS anatomy.	13
2	Token tree.	14
3	Final AST from parser.	14
4	Macro Expansion.	15
5	Term tree.	15
6	Broken unless macro.	16
7	Macro expansion of the broken *unless* macro	17
8	Pre-processing [Approach 1.]	20
9	Calling unless macro	20
10	"it" identifier correctly bound to $cond... in an anaphoric-if	21
11	Approach 2.Shows how we can use anaphoric-if to build an "unless" macro	23
12	Approach 2. Syntax parameter expansion	24
13	Approach 2. Design	24
14	Approach 2. context environment object loaded with macro definition	25

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 code 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 identifiers [3]. They only require lexical analysis; that is, they operate on the source text prior to any parsing, using simple substitution of tokenized character sequences for other tokenized character sequences according to user-defined rules. Consider this example [17]

```
#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 the C preprocessor results 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 introduces a new scope. The output of the compiled program is:

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

The solution is to give the macro's variables names that do not conflict with any variable in the current program:

```
#define INCI(i) {int INCIa=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;
 }
```

this solution produces the correct output:

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

The problem is solved for the current program, but this solution is not robust. This is where we require a hygienic macro system, which preserves the lexical scoping of all identifiers.

2. **Syntactic macro systems**, like those in the Lisp and Scheme [12] 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 [3]. Below is an example [21] of a syntactic macro in scheme that swaps the values.

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

The 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 that uses the pattern variable matched.

```
(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 of (6 5). Racket's [14] pattern-based macros automatically maintain lexical scope, so macro implementors can reason about variable references in macros and macro uses in the same way as for functions and function calls.

1.2 Hygiene

Hygienic macros are macros whose expansion does not cause the accidental capture of identifiers introduced by the macro expander. Hygiene prevents variable names inside the macros from clashing with the variables in the surrounding code. Hygienic macro systems are a feature of programming languages such as Scheme. Consider the following Scheme macro for "or" [19]

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

We can call the above macro as shown below:

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

The macro call is expanded to:

```
(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, the 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.

There are occasions when traditional hygienic binding is insufficient: one example is the "anaphoric if condition" (a version of the if-then-else construct that introduces an anaphor "*it*," which is bound to the result of the test clause), where expanding the macro definition at compile time may deliberately 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.

```
(aif (big-long-calculation)
     (foo it) ;;'it' is the result of big-long-calculation
     #f)
```

when the condition is true, an *it* identifier is automatically created and set to the value of the condition.

1.3 Syntax Parameters

Syntax parameters are a mechanism for rebinding a macro definition within 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 are not possible in other languages. With syntax parameters, instead of introducing the binding unhygienically each time, we instead create one binding for the keyword, which we can then adjust later when we want the keyword to have a different meaning. As no new bindings are introduced hygiene is preserved. Looping macros are a common example of syntax parameter that introduces a “break" or “abort" function [1].

The below example shows how to define a syntax parameter in Racket [11]:

```
#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 within 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 transformation happens at compile time 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.

Shown below is an example of **Break** keyword implementation using a syntax parameter [16]

```
(define-syntax-parameter break
    (constant-syntax-func
    #'(error "Error - must be used in breakable")))
```

```
(define-syntax (breakable stx)
    (syntax-parse stx
        [(_ body ...+)
        #'(call/cc
            (lambda (cc)
                (syntax-parameterize
                ([break (constant-syntax-func #'(cc))])
                    body ...)))]))

(breakable
    (displayln "beginning")
    (displayln "middle")
    (break)
    (displayln "end"))
```

This creates exactly the behavior that we desire, calling *break* outside the form raises an error with an appropriate message explaining its proper usage, and *breakable* forms are naturally supported with each break form only skipping one level, because each instance of *breakable* parameterizes break separately to a different continuation.

In this paper, I present the example of a macro that breaks hygienie in Sweet.JS and propose a solution taking inspiration from Scheme's syntax parameters. The problem and solution are discussed in up-coming chapters.

CHAPTER 2

Basics of Sweet.JS

Sweet.JS [6] is a hygienic macro compiler for JavaScript that takes JavaScript macros and produces normal JavaScript code that 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 compile time and produces the expanded source that can be run in any JavaScript environment.

2.1 Types 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 based macro the grammar is

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

The following macro defines swapping two values [20]:

```
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 and 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 maintain hygiene by renaming variables. Basically they track the scope of variables during expansion and rename them to maintain the correct scope and avoid accidental variable capture. 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 macros allow the macro author to use JavaScript code to procedurally create and manipulate the syntax. To define case macros, the grammar is

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

The following macro adds syntax for generating random numbers [20]:

```
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. "Letstx" which bind the pattern variable to syntax object.

Breaking hygiene using "case" macro is done by stealing the lexical context from syntax objects in the right place [6]. Consider *aif* the anaphoric if macro that binds its condition to the identifier *it* in the body as shown in the below example

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

This is a violation of hygiene because normally *it* should be bound to the surrounding environment but *aif* wants to capture *it*.

```
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 $it = [it];
  return #{
      // create an Immediately-Invoked Function expression that
   binds `$cond` to `$it`
      (function ($it) {
          if ($cond ...) {
          // all `it` identifiers in `$body` will now
          // be bound to `$it`
          $body ...
        }
      })($cond ...);
      }
  }
}
```

To do this we can create an *it* binding in the macro that has the lexical context within surrounding environment using *makeIdent* represent the string value as an identifier and *letstx*, which binds syntax object to pattern variable. The lexical context we want is actually found on the *aif* macro name itself [6].

2.2 Sweet.JS compilation process

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 [2]. The parser gives structure to unstructured source code. The

Figure 1: Sweet.JS anatomy.

lexer converts a character stream to a token stream and the parser converts the token stream into an abstract syntax tree (AST) according to a context free grammar [2]. The macro expander must sit between the lexer and the parser. Here the reader records sufficient history information in the form of token trees in order to decide how to parse the token, which is required to decide if a token is a divisor ("/") or a regular expression. In traditional JavaScript compilers, the parser and lexer are intertwined; rather than running 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 characters are tokenized accordingly [2] for example if the parser receive the characters "/x/" the lexer will result in a single token "/x/" otherwise lexer will result in individual tokens /,x,/ .

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.

The reader converts the string of tokens to a token tree. A token tree is similar to tokens produced by the standard esprima lexer [7], but with the critical difference that token trees match delimiters.

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

Figure 3: Final AST from parser.

The approach used in Sweet.JS is *enforestation*, first pioneered by Honu [4]. Enforestation extracts the sequence of terms produced by the reader to create a term

tree. Consider the following `let` example [2]

```
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.

Enforestation begins by loading the *let* macro, into the macro environment and converting the function declaration into a term tree as shown in Figure 4.

```
<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.

Figure 5: shows how the let example in figure 4 expanded to a term tree using the *var* keyword introduced by the rule macro.

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].

2.3 Current limitation of Sweet.JS

Although the benefits of hygienic macros are well established, there are occasions when traditional hygienic bindings are insufficient. The classic example is

```
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 ... // 'it' is not defined here, unfortunately
            }
        }
    }
}
unless (x) {
    // `it` is not bound!
    console.log(it)
}
```

Figure 6: Broken unless macro.

"anaphoric conditionals" where the value of the tested expression is available as an *it* binding. 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 as an anaphor (an expression referring to another expression).

```
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 of the broken *unless* macro

In Figure 6, I define the *"unless"* macro that executes code if the condition is false. If the condition is true, the code specified in the else clause is executed. Here we use an *anaphoric-if* condition, which introduces an anaphor *it* that should bind to the result of the test clause.

In Figure 7, on the macro expansion in line (8), the identifier `it` is not defined. Here we wish to introduce the identifiers deliberately breaking hygiene.

The same unhygienic macros are possible in Scheme as shown in the below example:

```
1 (define-syntax-rule (aif condition true-expr false-expr)
2  (let ([it condition])
3    (if it
4       true-expr
5       false-expr)))
6 aif #t (displayln it) (void))
7 ------------------------------------------------------------
8 it: undefined; //error
9 cannot reference an identifier before its definition
10 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 keyword to the value obtained by evaluating the transformer. The transformer provides the default expansion for the syntax parameter. Usually, you will just want to have the transformer throw a syntax error indicating that the keyword is supposed to be used in conjunction with another macro, as shown in the below example [11].

```
(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)))
```

The `"syntax-parameterize"` function adjusts the keyword to use the values obtained by evaluating their transformer in the expansion of the expression. Each keyword must be bound to a syntax-parameter.

```
(aif 10 (displayln it) (void))
```

Inside syntax-parameterize, *it* acts as an alias for *tmp*, results in 10 being displayed. The alias behavior is created by **make-rename-transformer**.

CHAPTER 3

Syntax parameters implementation for Sweet.JS

Syntax parameters are a mechanism for rebinding a macro definition within the dynamic extent of a macro expansion. With syntax parameters, instead of introducing the unhygienic binding each time we instead create a binding for the identifier, which we can adjust later when we want the identifier to have a different meaning as no new binding is introduced 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 Pre-processing [Approach 1]

In this approach, I tried to use the pre-processing approach similar to C, where I transform the code before the main compiler get a hold of it. ''`SyntaxParameter`'' replaces and binds the parameter with its mapped value in the particular defined macro scope. Here the macro transformation happens during the parse phase by matching the pattern. To define syntax parameters in Sweet.JS, the programmer provides a new keyword that looks something like this:

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

"parameter" is an identifier that is defined as a syntax parameter in the macro definition, *"Mapped to"* is the macro input variable that will be mapped to *"parameter,"* and *"Scope"* defines the context of the macro definition.

An example of its usage is Figure 8:

```
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: Pre-processing [Approach 1.]

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

Figure 9: Calling unless macro

Figure 10 shows the expanded code for the broken *"unless"* macro

```
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 correctly bound to $cond... in an anaphoric-if

As a example in Figure 10 shows, "`it`" is correctly bound to $cond... as desired. However this approach has certain disadvantages, since this will not allow users to define macros named "`SyntaxParameter`" 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 new syntax (thus a syntax transformer). To fix this we first need to implement some primitive functions that help us to create and manipulate the arbitrary compile time syntax transformation. Macros are compile time syntax transformations, 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 Syntax parameter processing at compile time [Approach 2]

The disadvantage with [Approach 1] is that it does not allow users to define macros named "SyntaxParameter". In this approach I defined syntax parameter that can be processed at compile time, the following paradigm are added in order to define syntax parameter in Sweet.JS.

1. **"syntaxparam,"** similar to define-syntax-parameter in Scheme, loads the primitive function in the compile time environment.

2. **"syntaxLocalValue,"** which loads the compile time primitive function from the environment within the dynamic extent of the macro expansion.

3. **"replaceSyntaxParam,"** which transforms the identifier with the compile time value from the environment returned by "syntaxLocalValue" within the defined scope of the macro.

In Figure 11, I have defined an anaphoric-if macro named as *aif*, where *syntaxLocalValue* defines an identifier *it* as a syntax parameter defined in the *aif* macro context, which also loads the macro definition for *it* from the context environment. Once the macro definition is loaded, *replaceSyntaxParam* replaces the identifier with the defined identifier definition during expansion of the macro.

```
    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 ...
                    }
                })

            }
        }
    }

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 11: Approach 2.Shows how we can use anaphoric-if to build an “unless” macro

```
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 12: Approach 2. Syntax parameter expansion

Figure 12 shows the expanded result of the "unless" macro defined in Figure 11, and *it* expands as expected. The source code is available in `Appendix A.1`.

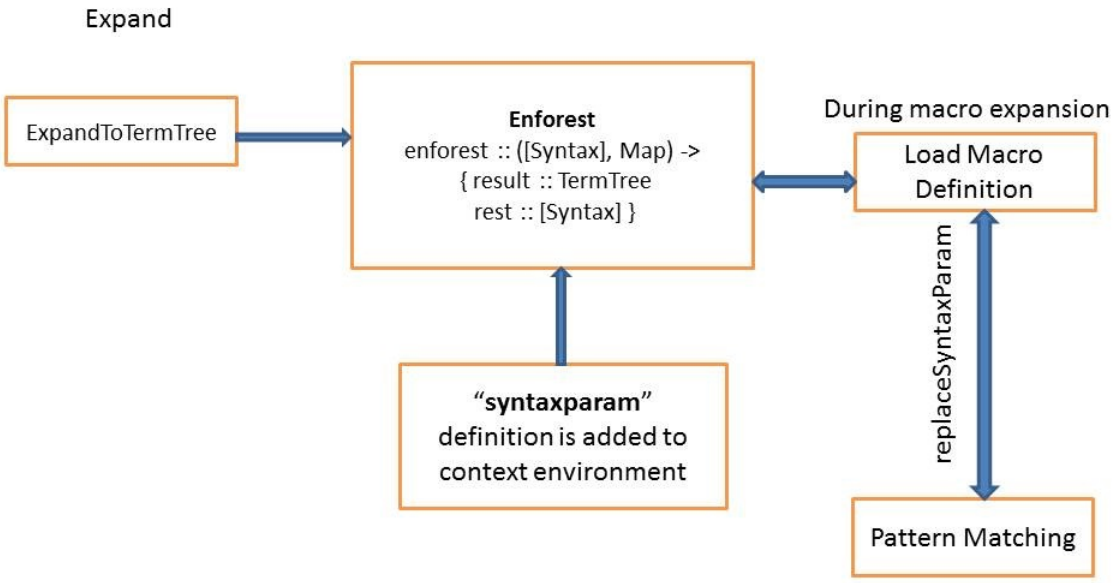

Figure 13: Approach 2. Design

The main entry point into the expander is the expand function, which is primarily responsible for handling hygiene. The *env* param is a mapping from identifiers to macro definitions, as shown in Figure 14 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.

The "expandToTermTree" function calls *enforest* repeatedly until the entire token tree has been converted into a term tree. In the enforest function, the "syntaxparam" definition loads the macro to the context *env* map [enforest :: [Syntax],Map)]. When a macro call is invoked the macro expander loads the macro definition from *env* in the "loadmacrodef" function, the "replaceSyntaxParam" function binds the identifier with the value of the loaded macro definition during pattern matching.

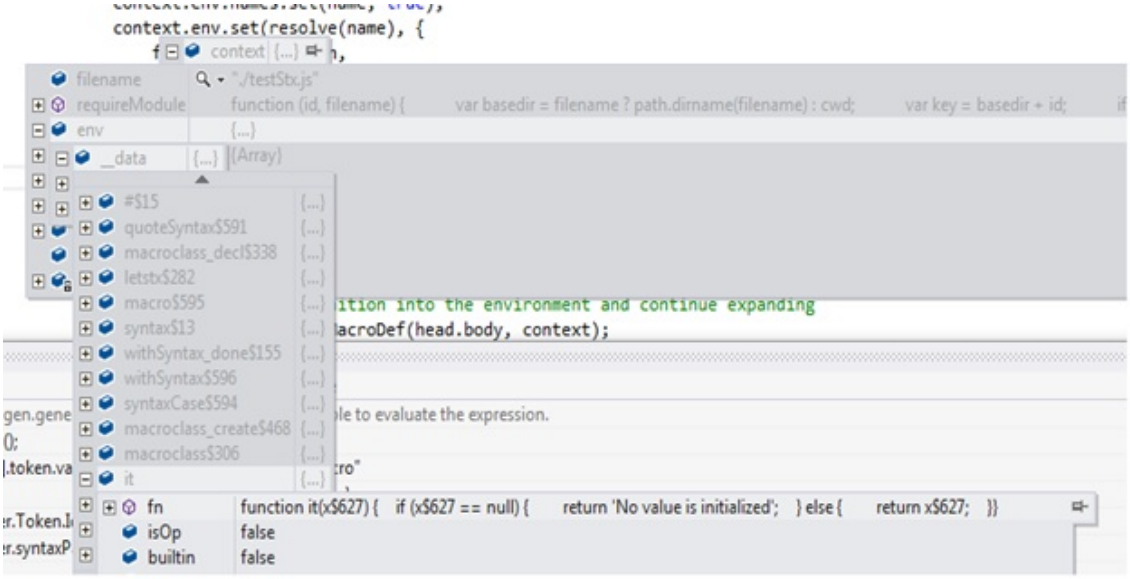

Figure 14: Approach 2. context environment object loaded with macro definition

Conclusion

Syntax parameters are an outstanding instrument for rebinding a macro definition within the dynamic context of a macro expansions thereby introducing implicit identifiers in a hygienic fashion. With syntax parameters, instead of introducing the binding unhygienically each time, we instead create one binding for the keyword, which we can then adjust later when we want the keyword to have a different meaning. As no new bindings are introduced, hygiene is preserved. In my implementation, I define "syntaxparam," which defines and binds the syntax parameter part of the compiler; "syntaxLocalValue," which pulls the syntax parameter definition in the defined scope; and "replaceSyntaxParam," which expands the syntax parameter macro definition defined within the macro body using "syntaxLocalValue".

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. (2014, 8 21). Retrieved from Sweet.JS Documentation,
http://sweetjs.org/doc/main/sweet.html

[7] The Esprima JavaScript Parser. (*visited on*2014,11 11)
http://esprima.org/

[8] T. Disney, N. Faubion, D. Herman, and C. Flanagan. The Sweet.JS Appendix. (2014, 8 21)
https://github.com/mozilla/sweet.js/blob/master/doc/dls2014/sweetjs-appendix.pdf

[9] R. Kent Dybvig , Robert Hieb , Carl Bruggeman,
Syntactic abstraction in Scheme, Lisp and Symbolic Computation, v.5 n.4, p.295-326, Dec. 1992.

[10] E. Kohlbeker, D.P. Friedman, M.Felleisen, and B. Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, 151-161, New York, NY, USA, 1986. ACM.*

[11] Greg Hendershott. Fear of Macros(*visited on* 2014, 7 11)
http://www.greghendershott.com/fear-of-macros/index.html

[12] R. Kent Dybvig. *The Scheme Programming Language. The MIT Press, fourth edition, 2009b.*

[13] Scheme Syntax Parameters
http://download.plt-scheme.org/doc/html/reference/stxparam.html

[14] Racket Syntax Transformers (*visited on* 2014, 7 11)
http://docs.racket-lang.org/reference/stxtrans.html

[15] Racket Syntax Parameters. (*visited on* 2014, 7 11)
http://docs.racket-lang.org/reference/stxparam.html

[16] Syntax Parameters. (*visited on* 2014, 7 26)
http://codepen.io/Universalist/blog/continuations-and-creating-a-break-keyword

[17] Hygienic macro. (*visited on* 2015, 5 1)
http://en.wikipedia.org/wiki/Hygienic_macro

[18] Macro (computer science). (*visited on* 2015, 3 13)
http://en.wikipedia.org/wiki/Macro_%28computer_science%29

[19] Macro Hygiene by Eric Holk. (*updated on* 2013, 6 17)
http://blog.theincredibleholk.org/blog/2013/06/17/what-is-macro-hygiene/

[20] Stop Writing JavaScript Compilers Make Macros Instead By James Long (*visited on* 2014, 10 17)
http://jlongster.com/Stop-Writing-JavaScript-Compilers--Make-Macros-Instead

[21] Pattern Based Macros (*visited on* 2014, 10 17)
http://docs.racket-lang.org/guide/pattern-macros.html

[22] Code base: Github
https://github.com/VimalKumarS/cs297

APPENDIX

Code base

A.1 Syntax parameter processing at compile time [Code base]

```
// enforest the tokens, returns an object with the `result`
//  TermTree and the uninterpreted `rest` of the syntax
function enforest(toks, context, prevStx, prevTerms) {
    .....
//Syntax param syntaxparam ToDo: add the stx as syntax
//like unwrapSyntax(head.keyword)
    if(resolve(head) === 'syntaxparam' &&
      rest[1].token.value==="=>"){
        head.syntaxParamName=rest[0]
        head.body=rest[2].token.inner
        head.isSyntaxParam=true
       //   parser.Token.Keyword
        return {
                  result: head,
                  rest: rest.slice(3),
                  opCtx:opCtx
                  };
        }
  .......
  }
```

```
// given the syntax for a macro, produce a macro transformer
// (Macro) -> (([...CSyntax]) -> ReadTree)

function loadMacroDef(body, context) {
    .....
     replaceSyntaxParam:syn.makeSyntaxParam, // Added
     syntaxLocalValue:function(id,stx){
          return syn.syntaxLocalValue(id,stx,context) //Added
                          }
    ....
   }
```

```
// similar to 'parse1' in the honu paper
// ([Syntax], Map) -> {terms: [TermTree], env: Map}
function expandToTermTree(stx, context) {
     .....
      //Add syntax param context
    if(head.isSyntaxParam){
       var name= head.syntaxParamName
        // head.body
      macroDefinition = loadMacroDef(head.body, context)

      context.env.names.set(name.token.value, true);
      context.env.set(resolve(name), {
            fn: macroDefinition,
            isOp: false,
            builtin: false,
            fullName: name
            });

        syntaxParamPrimitiveFn[resolve(name)] =
                  [syn.makeIdent(macroDefinition(),head)];
        continue;
        }
            //
     .....
      }
```

```
// given the macroBody
//(list of Pattern syntax objects) and the
// environment (a mapping of patterns to syntax)
//return the body with the
// appropriate patterns replaced with
//their value in the environment
// pattern.js
function transcribe(macroBody, macroNameStx, env) {
      .....
  if(parser.syntaxParameter[macroNameStx.token.value]
          != undefined )
  {
  newBody.token.inner=syntaxParamMatch(newBody.token.inner,
        parser.syntaxParameter[macroNameStx.token.value],env);
  }
   acc.push(newBody);
   return acc;
      ....

 }
```

```
// pattern.js
function syntaxParamMatch(_inner,paramValue,env){
   if(paramValue != undefined){
       _.each(_inner, function(inner,key) {
         if(inner.token.value == "()")
         {
           inner.token.inner=
           syntaxParamMatch(inner.token.inner,paramValue,env)
         }
         else if(inner.token.value == paramValue.param)
         {
           var last=_.last(_inner,_inner.length-key-1);
           _inner=_.first(_inner,key);
            push.apply(_inner,paramValue.value);
            push.apply(_inner,last);
          }
            return _inner;
       });
    }

   return _inner;

}
```

A.2 Syntax parameter pre-processing [Code base]

```
1
2  // Code Added to Parser.JS
3      // (Str) -> [...CSyntax]
4      function read(code) {
5
6          ........
7          while (index < length || readtables.peekQueued()) {
8             var obj=readToken(tokenTree, false, false);
9             if(obj.value=="defineSyntaxParameter")
10              {
11                 // obj=readToken(tokenTree, false, false);
12                 //defineSyntaxParameter[obj.inner[0].value]=obj.inner.
    splice(1,obj.inner.length);
13                 obj.value="macro"
14                // continue;
15              }
16            if (obj.value=="{}" && obj.inner != undefined) // looking for
17              {
18                fetchSyntaxParameter(obj);
19
20              }
21
22             tokenTree.push(obj);
23         }
24     ........
25     }
```

```
// Code Added to Parser.JS
// This function pull the syntax parameter during parsing
// Create an object 'syntaxParameterParse' to keep track
// of the syntax parameter defined in the context
   function fetchSyntaxParameter(obj)
     {

         for (var i in obj.inner)
                {
                     var paramObj=[];
                     var bflag=false;
                   // obj.inner[i]
                    if (obj.inner[i].value =="{}")
                    {
                       var tempParameter=obj.inner[i].inner.slice();

                        for (var j in tempParameter)
                       {
                           if (tempParameter[j].value ==" SyntaxParameter")
                            {
                                  // console.log("found");
                                   paramObj.push(tempParameter[j])
                                obj.inner[i].inner.shift();
                                bflag=true;

                            }
                           else if(tempParameter[j].value =="()" && tempParameter[j].inner != undefined && bflag)
                               {
                                    var k=0;
```

```
30                                      while (k<7) // fetch the syntax
    parameter
31                                          {
32                                              if(obj.inner[i].inner[0].
    inner[0].value !=",")
33                                              {
34                                              paramObj.push(obj.inner[i].
    inner[0].inner[0])
35                                              }
36                                              obj.inner[i].inner[0].inner.
    shift();
37                                               k++;
38                                      }
39
40                                      break;
41                                  }
42                              else
43                                  {
44                                      break;
45                                  }
46                              }
47
48
49
50                      }
51                      if(paramObj.length>0)
52                          {
53                              // create the syntaxPatameter based on the
    scope defined in the macro
54                          syntaxParameterParse[paramObj[paramObj.length
    -1].value]=createSyntaxParamArray(paramObj);
```

```
55
56
57                      //remove the syntax parameter () and push the
    value one level up
58                       for (var l = 0, len = obj.inner[i].inner[0].
    inner.length; l < len; ++l) {
59                           obj.inner[i].inner.splice(l+1,0,obj.inner[i
    ].inner[0].inner[l]);
60                       }
61
62                      obj.inner[i].inner.shift();
63                       iNumParam++;
64                     }
65              }
66
67    }
68    function createSyntaxParamArray(_paramObj)
69    {
70          return {
71              "param" : _paramObj[1],
72               "value" : [_paramObj[2],_paramObj[3]]   // Todo: Need to
    add te remaining array element
73               }
74    }
```

```
//Code added to pattern.js
//function add the syntax parameter with the literal definition during expansion
    function syntaxParamMatchParse(_inner,paramValue,env)
    {
        _.each(_inner, function(inner,key) {
                        if(inner.token.value == "()")
                            {
                                syntaxParamMatchParse(inner.token.inner, paramValue,env)
                            }
                        else if(inner.token.value == paramValue.param.value)
                            {
                                _inner[key];
                                push.apply(_inner, joinRepeatedMatch(env[paramValue.value[0].value].match, " "))
                                _inner.splice(key,1);
                                //return inner
                            }
                });


    }
```