Adding Syntax Parameter to 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 Parameter to Sweet.js macro library for JavaScript

by

Vimal Kumar

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2015

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

ABSTRACT

Adding Syntax Parameter to Sweet.js macro library for JavaScript

by Vimal Kumar

Macros have a long history in computing, Mozilla Sweet.js provides a way for developer to enrich the JavaScript by adding new syntax to the language through the use of macro. Sweet.js provide the possibility to define the hygienic macros inspired by Scheme. In this paper I present the implementation of the Syntax-Parameter feature to SweetJS library. Syntax parameter is a mechanism for rebinding a macro definition within the dynamic extent of a macro expansion. In implementation I defined the "syntaxparam" which define and bind the syntax parameter part of the compiler, "syntaxLocalValue" which pull the syntax parameter definition in the defined scope and "replaceSyntaxParamTransform" which transform to the identifier within the macro body using syntaxLocalValue.

ACKNOWLEDGMENTS

I have taken efforts in this project. However, it 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.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 What is macro?

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

There are two type of macro system

1. **Lexical macro systems** , called lexical macro system, such as C preprocessor, "Preprocessor" generally means a tool that transform the codes before the main compiler get hold of it. Lexical macros are ignorant of the grammar of core programming language and this sometime result in ill-formed program and result in accidental capture of the identifier. 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;
}
```

2. **Syntatic macro systems** , like Lisp and Scheme programming language. these macro system are aware of the grammar of core programming language they transform the syntax tree according to a number of user-defined rules. Rule can be written in the same programming language as the program, or in other language that rely on fully external language to define the transformation, such as the XSLT preprocessor for XML. Example

```
(define-syntax swap!
        (syntax-rules ()
((_ a b)
 (let ((value a))
 (set! a b)
 (set! b value)))))
```

1.2 What Sweet.JS bring for us?

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

Sweet.js provide the two way to define a macro: simple pattern based rule macros work by matching a syntax pattern generating the new pattern based on the template and the more powerful procedural case macros allow you to manipulate syntax. Example

```
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 How to break hygiene?

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

There are occasions when traditional hygienic binding are insufficient example, "anaphoric if condition" where while expanding the macro definition at compile time may introduce new variable binding, these new binding can end up capturing variables in your own code. That is new binding might shadow a variable which 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))
```

expand to

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

This program evaluate to #f, which is not the desired output, on expanding the macro the binding t is shadowed to #f. This issue is being resolved in Scheme using syntax-parameter.**Syntax-parameter** are a mechanism for rebinding a macro definition with in the dynamic extent of a macro expansion.

The same problem i observed with Sweet.JS library, in this paper i present the example of macro where it break Sweet.JS hygienic macro and proposed the solution taking inspiration from the Scheme's syntax parameter. Problem and solution discussed in up-coming chapter.

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?

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

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

   Lets write the macro that define 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.

it 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 this

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

It looks a little ugly, but 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 macro author to use javascript code to procedurally create and manipulate the syntax. To define case base macro

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

   Example

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


rand x;
```

   Expand To

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

   case is run at expand-time and you use #{} to create "templates" that construct code just like the rule in the other macros.

2.2 How Sweet.JS work?

JavaScript macro system, sweet.js include a separate reader that converts a sequence of tokens into a sequence of token trees,analogous to s-expression in scheme,without feedback from the parser.

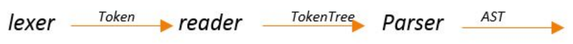


Figure 1: Sweet.JS anatomy.

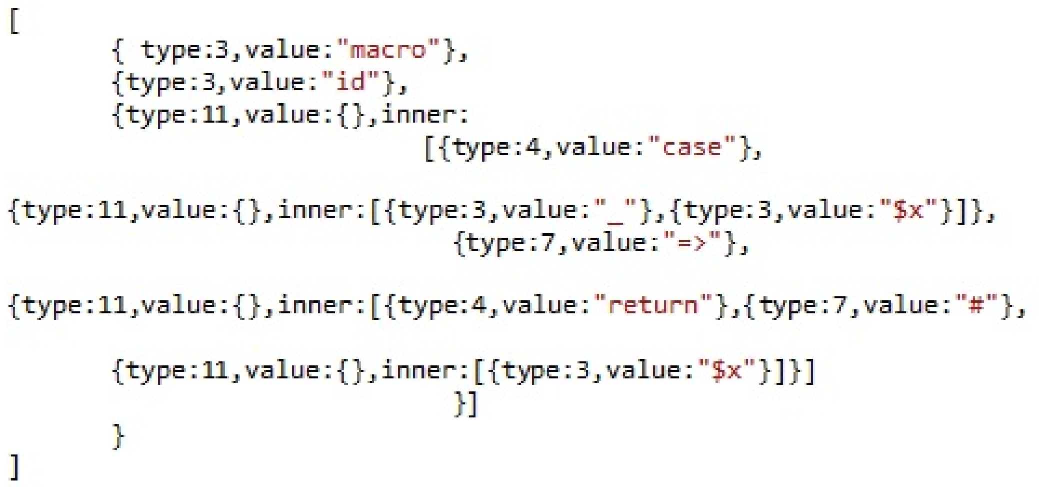
Parser give structure to unstructured source code. lexer which converts a character stream to a token stream and a parser which converts the token stream into an AST according to a context free grammar [2].System which expand the macro definition must sit in between lexer and parser.Here reader records sufficiently history information in the form of token tree in order to decide how to parse the token, this is required to decide if token is a divisor or a regular expression.

Example

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

Reader convert string of token to token tree,

The approach used in Sweet.JS named as Enforestation, was pioneered by Honu [4]. Enforestation means transformation, it extract the sequence of terms produced by the reader to create a term tree. A term tree is a kind of proto-AST that represent the partial parse 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.



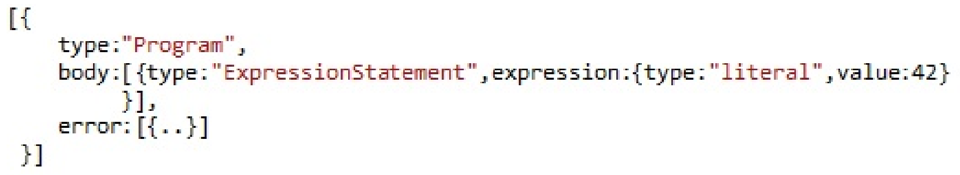
```
[
        { type:3,value:"macro"},
        {type:3,value:"id"},
        {type:11,value:{},inner:
                                [{type:4,value:"case"},

{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.

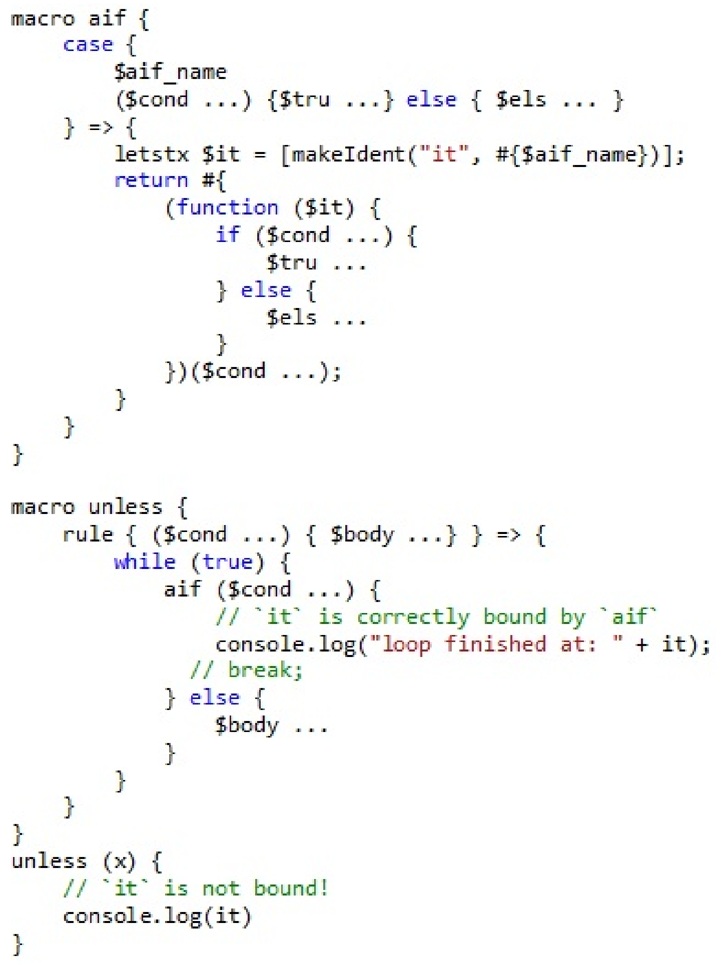


```
[{
    type:"Program",
    body:[{type:"ExpressionStatement",expression:{type:"literal",value:42}
        }],
    error:[{..}]
 }]
```

Figure 3: Final AST from parser.

2.3 Problem trying to solve?

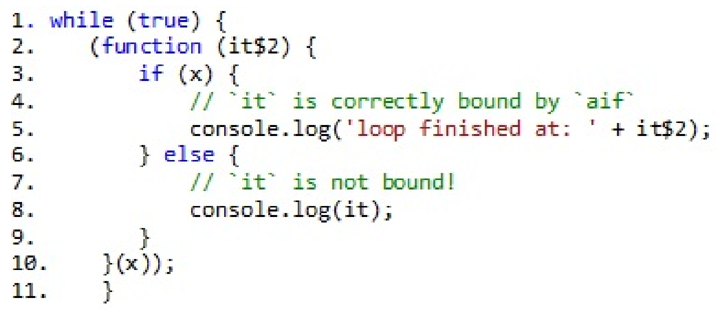
Although the benefits of hygienic macros are well established, there are occasions when traditional hygienic bindings are insufficient. let's understand this via example "anaphoric conditionals" where value of the tested expression is available as an 'it' bindings.That means when the condition is true, an "it" identifier is automatically created and set to the value of the condition.



```
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 4: Break hygienic.



```
while (true) {
   (function (it$2) {
       if (x) {
           // `it` is correctly bound by `aif`
           console.log('loop finished at: ' + it$2);
       } else {
           // `it` is not bound!
           console.log(it);
       }
    }(x));
    }
```

Figure 5: Macro expansion.

In above macro expansion, at line (8) identifier 'it' is not defined.Here we wish to introduce the identifiers as is, unhygienically. Same unhygienic macros are observed in Scheme as shown in the below example, mistake is if our new syntax introduces a variable that accidentally 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
```

Good way is with a syntax parameter, as shown below inside the syntax-parameterize, "it" acts as an alias for tmp. The alias behavior is created by make-rename-transformer.**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.

```
(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)))
```

If we try to use it outside of an aif form, and it isnâĂŹt otherwise defined, we get an error

```
(displayln it)


error->it: can only be used inside aif
```

But we can still define it as a normal variable in local definition contexts like:

```
(let ([it 10])
  it)


output: 10
```

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, instead create binding for the keyword, which we can then 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 dynamic binding mechanism we have at run time, except that the dynamic binding only occurs during macro expansion.

3.1 Appraoch 1

3.2 Appraoch 2

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`