What Macros Are and How to Write Correct Ones

Brian Goslinga
University of Minnesota, Morris
600 E. 4th Street
Morris, Minnesota 56267
gosli008@morris.umn.edu

ABSTRACT

Macros are a powerful programming construct found in some programming languages. Macros can be thought of a way to define an abbreviation for some source code by providing a program that will take the abbreviated source code and return the unabbreviated version of it. In essence, macros enable the extension of the programming language by the programmer, thus allowing for compact programs.

Although very powerful, macros can introduce subtle and hard to find errors in programs if the macro is not written correctly. Some programming languages provide a hygienic macro system to ensure that macros written in that programming language do not result in these sort of errors. In a programming language with a hygienic macro system, macros become a reliable part of the language.

Macros in C, Common Lisp, and Scheme will be explored, focusing on issues relating to the correctness of macros in each language. To demonstrate macros solving a problem, a Project Euler problem will be solved with the help of macros. Finally, some current work relating to hygienic macros in the Racket programming language will be discussed.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Macros, hygienic macros, macro hygiene, programming language design

1. INTRODUCTION

1.1 Macros

Code that is *declarative* states what it does, and not how it does it. Declarative code is easier to read, write, and understand because the intent is not obscured by implementation details. This is the underlying idea in high-level languages such as Java and Scheme. There arise situations in most high-level languages where code falls short of this ideal by having redundancy and complexity not inherent to the problem being solved. Common abstraction mechanisms, such as functions and objects, are often unable to solve the problem [6]. Languages like Scheme provide macros to handle these situations. By using macros when appropriate, code can be declarative, and thus easier to read, write, and understand.

Macros can be thought of as providing an abbreviation for common source code in a program. In English, an abbreviation (UMM) only makes sense if one knows how to expand it into what it stands for (University of Minnesota, Morris). Likewise, macros undergo *macro expansion* to derive their meaning. Just as abbreviations are text-to-text transformations, macros are source-to-source transformations [2].

As an example, one can define additional control-flow constructs to a language using macros. `unless` is such a construct, and is equivalent to an `if` with a negated conditional. To add support for `unless`, we just write a macro that will turn expressions of the form `unless <condition>: <code>` into `if not <condition>: <code>`. The compiler can compile `unless isEmpty(file): readData(file)` by expanding the macro into `if not isEmpty(file): readData(file)`. `unless` cannot be defined as a function because a function would have its arguments evaluated before being called, and so it would try to read data from the file even if it was empty. This behavior is called *call-by-value* evaluation order, and is used by the vast majority of programming languages. This also shows that macros can do things that functions cannot.

Macros create specialized sub-languages, allowing for the succinct expression of otherwise verbose concepts in the programming language. Using macros, the programmer can create, encapsulate, and reuse entire sub-languages in their programming language [3].

1.2 What Can Go Wrong?

When a macro is used, it produces source code to be used in place of the macro call. The macro must be well written or else the source code produced may not interact correctly with the environment in which it was called. For instance, it may introduce a variable whose name clashes with another variable, or it might cause some user code to be unintentionally evaluated multiple times. These can lead to errors that are hard to track down because their cause is obscured by the macro. The first situation only occurs if the programmer happens to use a particular variable name, and so can be very hard to detect. A macro that does not suffer from these flaws is called *hygienic*. Hygienic macros preserve the

```
(define f
  (lambda (x)
    (let ((y (* x 3)))
      (if (< y 10)
          (* y y)))))
```

Figure 1: A function in Scheme

behavior of the program no matter where they are used in the program.

A language with a hygienic macro system provides *hygienic macro expansion*, a method of maintaining the relationship between a variable's name and its value, respecting the structure of expressions in the process [2]. One does not need a hygienic macro system to write hygienic macros. However, a hygienic macro system provides guarantees about hygiene that a non-hygienic macro system cannot, and so greatly improves the reliability of macros.

1.3 Overview of Lisp-like languages

Multiple Lisp-like languages are used in this paper. The syntax of Lisp-like languages are very similar to each other, so this section will focus specifically on Scheme. Scheme is a *functional* programming language, meaning that programs are based on a composition of functions. In Scheme, all source code is comprised of lists. A list in Scheme is of the form `(a b c)`, with the empty list being `()`. When a list, say `(f a b c)`, is evaluated, one of three things happen: if `f` is a *special form* (such as `if`), the list is evaluated using special rules. If `f` is a macro, the macro produces a form to be used in its place. If neither of these are true, then `f` is evaluated as a function, and the remainder of the list is evaluated and passed to the function as arguments.

An anonymous function is introduced in Scheme by `lambda`. `lambda` takes a list of argument names, followed by some body code. Each argument name will be bound to the corresponding argument (the first to the first name, the second to the second, etc.) when executing the body.

Local variables are introduced using `let`. `let` takes a list of variable-value pairs and some body code. The variables from the variable-value pairs assume the values from the pairs when executing the body.

Figure 1 provides an example of defining a function in Scheme. This code defines `f` to be a function of one argument `x`, introducing `y` to be three times `x`, and returns `y` if `y` is less than 10, otherwise it returns nothing. It is important to note that the second argument to `if`, in this case `(* y y)`, is only evaluated when the condition evaluates to true.

2. MACROS IN C

In an *imperative* programming language, programs are structured as a sequence of commands that update memory. C is an example of an imperative language. It has macros that are based on substituting text in the source code with other text. The macro language in C is weak, only having a few constructs available. Due to certain limitations in C and the weak macro language, it can be difficult or even impossible to write a hygienic macro in C. Due to this, macros in C have gained a bad reputation and it is recommended that they should be avoided if possible. We use C macros to illustrate problems with unhygienic macros.

In C, a macro is defined using `#define`. In Figure 2,

```
#include <stdio.h>

#define SQ1(x) x*x
#define SQ2(x) ((x)*(x))

int main(int argc, char* argv[]) {
  printf("%d\n", SQ1(2+3));
  printf("%d\n", SQ2(2+3));
  int a = 4;
  printf("%d\n", SQ2(++a));
  return 0;
}
```

Figure 2: Macro usage in C

`#define SQ1(x) x*x` defines `SQ1` to be a macro of one argument `x`. Upon macro expansion, the argument to the macro replaces `x` in `x*x` to get the macro expansion. For instance, `SQ1(5)` would macro expand to `5*5`.

`SQ1` is the naive way to write a squaring macro: just multiply the input by itself. However, the first `printf` prints `11` instead of the expected result `25`. The reason for this is that `SQ1(2+3)` expands to `2+3*2+3`, which results in 11.

`SQ2` introduces parentheses to fix the problem. `SQ2(2+3)` expands to `((2+3)*(2+3))`, so second `printf` prints `25` as desired. Parentheses allow C macros to acheive some level of hygiene [2].

Unfortunately, `SQ2` is still incorrect because the argument of the macro is evaluated twice. This was not a problem in the above example because 2+3 does not have side-effects, but consider the third `printf` call. Here `SQ2(++a)` expands to `((++a)*(++a))`. To allow extra optimization, this expression is undefined in C [1]:

> Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression.

This means that the expression `((++a)*(++a))` invokes undefined behavior because the value of `a` is modified twice in the expression. When the code is compiled with the compiler GCC, the compiled code increments `a`, then increments `a` again, and then multiplies `a` by `a` to produce the result 36. When the compiler Clang compiles this code, the compiled code increments `a`, copies `a` to `tmp`, increments `a`, and then multiplies `a` by `tmp` to get 30. Because the expression is undefined, both results are valid. Thus because of the unhygienic nature of the macro, the value of `SQ2(++a)` depends on the compiler used.

This macro could be fixed by storing the value of the argument to a local variable, and then squaring that (the equivalent of `(let ((tmp x)) (* tmp tmp))` in Scheme). Unfortunately, C does not have a construct similar to `let` in the needed ways [2], and so double evaluation is one of the problems that plagues C macros.

In this case, a squaring function could have been used instead of a macro to ensure correct behavior. However, not all macros can be implemented as functions: functions evaluate their arguments before the call, and so functions cannot provide features such as conditional evaluation. The result is that macros should be used carefully and sparingly in C.

```
(defmacro sq1 (x) `(* ,x ,x))
(defmacro sq2 (x) `(let ((tmp ,x)) (* tmp tmp)))
(defmacro dosum1 (var n body)
  `(let ((sum 0))
     (dotimes (,var ,n)
        (incf sum ,body))
     sum))
(defmacro dosum2 (var n body)
   (let ((sum (gensym)))
     `(let ((,sum 0))
        (dotimes (,var ,n)
          (incf ,sum ,body))
        ,sum)))
(defvar sum 5)
(defvar x 4)
(print (sq1 (+ 2 3)))
(print (sq2 (incf x)))
(print (dosum1 n 2 sum))
(print (dosum2 n 2 sum))
```

Figure 3: Macro usage in Common Lisp

3. MACROS IN COMMON LISP

The family of Lisp programming languages are well known for their excellent macro facilities [5]. In the Common Lisp language, macros are full-fledged elements of the language, and have the entire language at their disposal. In Common Lisp, all source code is represented as lists, and the macros operate on them directly, making macros very powerful. While the macro system is unhygienic, utilities to provide hygiene in macros exist. However, they have to be manually used.

`let` in Common Lisp is the same as Scheme. `dotimes` is a keyword that is similar to a `for` loop in Java. `incf` is a keyword to increment a variable by a value[1]. `defvar` is the keyword in Common Lisp for introducing variables. Common Lisp has a facility for creating lists (or code) from a template: an expression preceded by a ` character is treated as a literal, except for the expressions inside that are preceded by a `,` character, which are evaluated. This is frequently used in macros. Thus `(let ((x 5)) `(* ,x ,(+ x 3)))` evaluates to the expression `(* 5 8)`.

The code in Figure 3 begins with squaring macros. Due to the nested nature of Lisp code, the simple solution (`sq1`) works as intended in the first print, but it would be incorrect at the second invocation as it evaluates its argument twice, resulting in a duplication of side-effects. `sq2` stores the value of the argument in `tmp` to ensure double evaluation does not happen, and so it works correctly. As a result, after the call `x` is 5, indicating the side effect happened only once.

In the case of `sq2`, the `tmp` variable that is introduced is harmless as no user code is evaluated inside the `let`. However, this is not true in general; `dosum1` is an example of a macro that evaluates user code inside a `let` that it creates. The third print should print out `10`, but instead it produces `0` since the `sum` variable introduced by the macro is shadowing the `sum` variable that is visible to the user of the macro. `dosum2` resolves this issue and so it is hygienic (assuming the user expects the code in the body to be evaluated multiple times if $n > 1$). The `gensym` function in Common Lisp returns a symbol different from all names that are already used in the program. Using `gensym`, the macro can internally use a variable name that is guaranteed to not conflict with any user code.

[1]Both `dotimes` and `incf` are themselves macros in Common Lisp, but their implementation is outside the scope of this work.

Pattern	Matches
`dolists`	`dolists`
`#:when`	`#:when`
`test`	`(odd? a)`
`clauses ...`	`(c (range a))`
`body`	`(+ a c)`
Template	**Replacement**
`test`	`(odd? a)`
`dolists`	`dolists`
`clauses ...`	`(c (range a))`
`body`	`(+ a c)`

Figure 4: Matchings and replacements

4. MACROS IN SCHEME

Macros in Scheme are similar to Common Lisp macros, but there are built in facilities for hygienic macros. In particular, Scheme was the first programming language to support hygienic macros [8]. A macro in Scheme is defined by `(define-syntax name expander)`, where `name` is the name of the macro and `expander` is evaluated to get the expansion function (a function that takes the macro call and returns the new source code). `syntax-rules` (see Section 11.19 of [8]) creates an expander function, and is one of the components of the hygienic macro system. `syntax-rules` is an implementation of Macro-By-Example (see [6]), meaning the macro is defined in terms of example usages of the macro.

We use a (simplified) subset of `syntax-rules`:

```
(syntax-rules (rule template)*)
```

with the star meaning that `(rule template)` can be repeated multiple times. The expander function created by `syntax-rules` will try to match the macro call against each `rule` in turn, and on the first successful match, fills in the corresponding `template` and returns it.

Consider the following rule-template pair:

```
((dolists ((#:when test) clauses ...) body)
 (if test (dolists (clauses ...) body)))
```

Suppose the macro call was

```
(dolists ((#:when (odd? a))
          (c (range a)))
  (+ a c))
```

In the process of trying the rule, several matches will be attempted. Figure 4 shows the various components of the rules and what they are trying to match against. In `syntax-rules`, lists are matched against lists. For brevity, these matches are omitted.

When the pattern is a symbol (this type of pattern is called a *pattern variable*) such as `dolists`, the pattern will match anything. Literals such as `#:when` will match only themselves. A symbol followed by `...` will match the rest of the list. In this case, all of the matches succeed. The next step is to fill in the template. As seen in Figure 4, the

```
(define euler-9
  (find-first
    ((a (range 1 1000))
     (b (range (+ a 1) (/ (- 1000 a) 2)))
     (#:let c (- 1000 a b))
     (#:when (= (* c c) (+ (* a a) (* b b)))))
    (* a b c)))

public int euler9() {
  for (int a : range(1, 1000)) {
    for (int b : range(a+1, (1000-a)/2)) {
      int c = 1000-a-b;
      if (c*c == a*a + b*b) {
         return a*b*c;
} } } }
```

Figure 5: The solution in Scheme and Java

symbols from the rule are replaced with what they matched in the rule.

When filling in the template, the case of `if` is special: it is not a symbol appearing in the rule. Since it was defined where the macro was defined, `if` will stay the same assuming that `if` does not mean something else when the macro is used. If `if` was not defined where the macro was defined, then it would have been renamed to a name that will not clash with any other variable name.

Filling in the template results in the macro expansion:

```
(if (odd? a) (dolists ((c (range a))) (+ a c)))
```

This expansion of the macro uses a nested `dolists` so the full macro expansion is not yet complete. The compiler will now expand the result of the macro expansion until no more macros remain.

5. EXTENDED MACRO EXAMPLE

5.1 A Project Euler problem using macros

A functional programming language like Scheme is a good choice for solving mathematical problems, and the Project Euler website provides many of them. Problem 9 [7] on Project Euler states

> A *Pythagorean triplet* is a set of three natural numbers, $a < b < c$, for which, $a^2 + b^2 = c^2$. For example, $3^2 + 4^2 = 9 + 16 = 25 = 5^2$. There exists exactly one Pythagorean triplet for which $a + b + c = 1000$. Find the product abc.

This problem can be solved by finding the first abc such that a, b, and c satisfy both constraints. We could do this directly in Scheme, but the resulting code would not be very declarative. We can make the solution code be declarative by writing a `find-first` macro, which will integrate the loops, conditionals, and variable definitions we would otherwise use. Figure 5 contains a solution to the problem using `find-first`. `range` is a function defined so that `(range a b)` returns a list of all integers x such that $a \leq x < b$.

Figure 5 also show a Java version of the Scheme solution. The `range` function in the Java code returns a list of integers in the same manner as the `range` Scheme function. To use `find-first` we have to write it, but once it is written it can be used anywhere.

```
(define-syntax dolists
  (syntax-rules
    ((dolists () body)
     body)
    ((dolists ((#:when test) clauses ...) body)
     (if test
       (dolists (clauses ...) body)))
    ((dolists ((#:let var val) clauses ...) body)
     (let ((var val))
       (dolists (clauses ...) body)))
    ((dolists ((var val) clauses ...) body)
     (for-each (lambda (var)
                 (dolists (clauses ...) body))
               val))))
(define-syntax find-first
  (syntax-rules
    ((find-first (clauses ...) val)
     (with-break
       (dolists (clauses ...)
         (break val))))))
```

Figure 6: `dolists` and `find-first`

5.2 Implementing `find-first`

We now need to implement `find-first`. Since we want `find-first` to be hygienic, we will define any needed macros with `syntax-rules` to get the hygiene for free. It is reasonable to have the macro produce a near direct Scheme translation of the Java code.

`find-first` is a bit complicated, so it would be useful to create a helper macro. We can write a `dolists` macro, see Figure 6, that will encapsulate all of the looping, conditional, and variable definition logic. If we had a `println` function that would print a value on a new line, then

```
(dolists ((a (range 1 10))
          (#:when (odd? a)))
  (println a))
```

would print the odd numbers between 1 and 9. `dolists` in effect creates a small language for expressing iteration. This ability is one of the benefits of macros.

The above example has two *clauses*, or forms that tell `dolists` what to do. The two clauses used above are `(a (range 1 10))` and `(#:when (odd? a))`; these clauses respectively tell `dolists` to loop over the list returned by `(range 1 10)` with `a` having the values in the list, and to proceed only when `(odd? a)` is true. The Project Euler solution used a third type of clause, `(#:let c (- 1000 a b))`. This clause instructs `dolists` to proceed with `c` being the value of `(- 1000 a b)`. This is equivalent to the first line of the inner loop in the Java version.

We will implement `dolists` recursively, so there are four cases to handle:

- There are no clauses (the base case of the macro).
- The first clause is a `#:when` clause.
- The first clause is a `#:let` clause.
- The first clause is a `for-each` clause.

These four cases correspond to the four rules in the definition of `dolists`, Figure 6.

The reason for the recursive implementation is the recursive structure in its macro expansion–the fact that `(dolists (a b) body)` is equivalent to `(dolists (a) (dolists (b)` `body))`. The usage of recursion is a very useful technique when writing macros.

`dolists` uses the `for-each` function to implement the for-each loop; this is useful as it eliminates the need to implement the iteration in the macro. Since `for-each` is a function, we have to wrap the body of the iteration in a function to block its evaluation until the appropriate time in the `for-each` call, and to introduce the variable that each value of the list will be bound to.

Defining `find-first` is relatively straight-forward with `dolists` in place. Using `with-break` and `break`[2], we can break out the `dolists` loops once the answer is found. This corresponds to the `return` statement in the Java version.

6. MACROS IN RACKET

Hygienic macro systems ensure that macros defined are hygienic, and so will behave correctly. However, there is more to correct macros than hygiene. Macros are an abstraction mechanism, but they are converted to a less abstract form before being compiled. If the macro was used incorrectly, the macro's expansion may not be valid code. In this case, the resulting error message will not be very helpful as it will be in terms of what the macro expanded into. The result is that the macro confuses and distracts the user because the abstraction it provides is broken. For a macro to be truly correct, it must also provide the correct response when its input is invalid.

Scheme's `syntax-rules` helps address this problem as the macro's usage must match one of several forms. Thus the macro programmer can specify some invariants that correct usage will have. However, `syntax-rules` is limited in what sort of invariants can be encoded in the macro's definition. Consider Figure 6. The rule for the for-each clause is

```
(dolists ((var val) clauses ...) body)
```

It is intended that `var` will ever only match a symbol, but this is not enforced by `syntax-rules` as `var` could match `(a b)`. If this happens, the macro expansion will contain `(lambda ((a b)) ...)`, which is not valid syntax for `lambda`. The result will be an error message that is removed from the context of the macro, and so can be difficult to understand. By using other methods to define the macro, it is possible to add validation code to the macro to ensure that `var` is actually a symbol, but this is an ad-hoc solution to the problem.

Additionally, there is another issue with `syntax-rules`: it is hard to create macros using `syntax-rules` that do not have a very homogeneous syntax. For example, keyword arguments are hard to handle, and keyword arguments that take a varying number of additional arguments are even more problematic. As a result of this, many macro writers are forced to simplify the grammar of the macro to make implementation feasible.

In [4], Ryan Culpepper and Matthias Felleisen introduce the macro system `syntax-parse`. `syntax-parse` is able to solve these issues with `syntax-rules`. `syntax-parse` is part of the Racket programming language, a dialect of Scheme. Users of Racket have confirmed that `syntax-parse` makes it easy to write macros with complex syntax. By using `syntax-parse`, the macro programmer can create much better abstractions than they could before, and the macros produced are very robust, declarative, and reusable.

```
(define-syntax-class binding
  #:description "binding pair"
  (pattern [var:identifier rhs:expr]))
(define-syntax-class distinct-bindings
  #:description "sequence of binding pairs"
  (pattern (b:binding ...)
    #:fail-when (check-duplicate #'(var ...))
                "duplicate variable name"
    #:with (var ...) #'(b.var ...)
    #:with (rhs ...) #'(b.rhs ...)))
(define-syntax (let stx)
  (syntax-parse stx
    [(let bs:distinct-bindings body:expr)
     #'((lambda (bs.var ...) body) bs.rhs ...)]))
```

Figure 7: Example definition of `let`

`syntax-parse` is similar to `syntax-rules`, but it provides powerful tools designed to simplify the verification of the macro usage, and improve the quality of the error messages produced by a syntax error. It does this by being able to detect a larger class of syntax errors before the macro is expanded into more primitive forms. This improves the quality of error messages, and increases the quality of the abstraction macros provide.

Additionally, `syntax-parse` is able to handle macros with less homogeneous syntax with an order of magnitude reduction of difficulty. As pointed out in [4], the `define-struct` macro in Racket had over a hundred lines of parsing code to handle the keyword arguments that `define-struct` can take. When `define-struct` was ported to `syntax-parse`, the number of lines required on parsing code was an order of magnitude less. This was possible because `syntax-parse` eliminated the need to write the complex parsing and validation code by hand.

`syntax-parse` derives much of its power by allowing pattern variables to be annotated with the type of expression that the pattern variable should match. These types are called *syntax classes*. For example, a pattern variable could be annotated with the syntax class of a symbol so that it would only match a symbol. The programmer can define their own syntax classes. A syntax class definition includes a description of the syntax, and can include extra validation code. Figure 7 shows the definition of two syntax classes that would be useful when defining `let` as a macro. In the figure, we see how a pattern variable is annotated with a syntax class. The syntax class of a symbol is `identifier`, and the syntax class of any expression is `expr`.

As seen in the figure, the description of the syntax class is introduced with the `#:description` keyword. Side conditions can be introduced with the `#:fail-when` keyword, which takes the validation code and the description of the error if the validation fails. `syntax-parse` also allows extra side conditions in the definition of the macro itself. This is useful if there is a side condition that only belongs to a single macro.

It is frequently the case that one wants access to the pattern variables that where defined in a syntax class. Using the dot syntax, one can gain access to these pattern variables. For example, in the definition of `let`, `bs.var`

[2]`with-break` and `break` are an abstraction we define over the facilities Scheme provides for jumping out of a function in the middle of its execution. Their implementation is beyond the scope of this paper.

```
> (let ([x 1] [x 2]) (h x))
let: duplicate variable name in: x
> (let ([(x y) (f 7)]) (g x y))
let: expected identifier in: (x y)
> (let (x 5) (add1 x))
let: expected binding pair in: x
> (let 17)
let: expected sequence of binding pairs in: 17
```

Figure 8: Errors `syntax-parse` produces

refers to the `var` pattern variable in the instance of the `distinct-bindings` syntax class that `bs` was matched to. Using `#:when`, a syntax class can export other pattern variables. `#:when` takes a pattern and a form to match. `#'expr` in Racket is the syntax for filling in the template `expr`. It is used in the definition of the syntax class `distinct-bindings` to get a copy of `b.var` and `b.rhs` for exporting the `var` and `rhs` pattern variables respectively.

`syntax-parse` also produces excellent error messages almost for free. Figure 8 shows errors messages produced by `syntax-parse` given the definition of `let` in Figure 7. The first error message shows the result when the validation code for the `#:fail-when` in `distinct-bindings` returns a true value. In this case it returned `x`, signifying that `x` was the culprit. The other three error messages show the result when the target of a pattern variable does not have the right syntax class.

The underlying idea in `syntax-parse` is that "error reporting should be based on documented concepts, not implementation details" [4]. In keeping of this philosophy, `syntax-parse` generates error messages from the description of the syntax classes used. A macro defined using `syntax-parse` knows there is a syntax error if no pattern completely matches the macro call. In the case that a simple pattern (such as a pattern variable) is the cause of the failure, `syntax-parse` will use the description of the simple pattern to generate the error message.

If the macro fails because of a compound pattern (such as a list pattern), reporting the description of the pattern is not sufficient. In general, the description of a compound pattern is little more than the pattern–an implementation detail. What `syntax-parse` will do in this case is search the stack of pattern matches to find what simple pattern this failure is contained in. `syntax-parse` will then generate an error message based off of this simple pattern instead. The third and fourth error messages in Figure 8 are the result of a compound pattern failing. As seen, `syntax-parse` found the nearest simple pattern and reported the corresponding error message.

The features given here are only a few of the features supported in `syntax-parse`. `syntax-parse` can guess what the user of a macro intended based off of how much each pattern matched. This allows a relevant error message to be produced. Other features that `syntax-parse` supports include syntax classes that take parameters and head patterns. Parametrized syntax classes allow for more abstraction in the definition of syntax classes. Head patterns are a variant of list patterns that allow for the easy definition of macros that take keyword arguments.

The result of all of the features in `syntax-parse` is that `syntax-parse` is a large step up from `syntax-rules`, just as `syntax-rules` was a large step up from unhygienic macros.

7. CONCLUSIONS

Macros are a very powerful construct that allow for succinct code by factoring out redundancies and implementation details. Using macros, one can create small sublanguages to allow for the precise formulation of domain-specific tasks. However, it is important for macros to be hygienic. If a macro is not hygienic, subtle and hard to find bugs can be introduced. A macro that is not hygienic is a broken abstraction that is nearly useless as one must contemplate the result of macro expansion to ensure the code is correct; defeating the purpose of having the macro in the first place.

C macros have many flaws when it comes to hygiene–even defining a simple squaring macro can be quite challenging. Due to this notorious lack of hygiene, macros in C should be avoided when possible. Common Lisp macros can be hygienic with some work, but one still has to define them carefully as hygiene is not provided automatically.

Scheme's hygienic macro system makes creating hygienic macros easy, as seen when we solved a Project Euler problem using macros. Languages like Scheme give their users great power by allowing them to extend the language when they feel that that it is the best choice.

Finally, we explored recent work in the field, namely the macro system `syntax-parse`. This system allows for even better macros to be easily written. These better macros handle syntax errors with grace and generate error messages that are much more comprehensible.

8. REFERENCES

[1] ISO/IEC JTC1/SC22/WG14 — C: Approved standards, 2010. [Online; accessed 15-November-2010].

[2] W. Clinger and J. Rees. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–162, New York, NY, USA, 1991. ACM.

[3] R. Culpepper and M. Felleisen. Debugging hygienic macros. *Sci. Comput. Program.*, 75(7):496–515, 2010.

[4] R. Culpepper and M. Felleisen. Fortifying macros. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 235–246, New York, NY, USA, 2010. ACM.

[5] E. Kohlbecker, 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*, pages 151–161, New York, NY, USA, 1986. ACM.

[6] E. E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 77–84, New York, NY, USA, 1987. ACM.

[7] Project Euler. Problem 9 — Project Euler, 2002. [Online; accessed 23-October-2010].

[8] M. Sperber, R. k. Dybvig, M. Flatt, A. Van straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language scheme. *J. Funct. Program.*, 19:1–301, August 2009.