What Macros Are and How to Write Correct Ones

Brian Goslinga

December 4, 2010

UMM Computer Science Senior Seminar

Abbreviations

English has abbreviations. Abbreviations allow for the compact representation of complex objects. Abbreviations allow us to chunk concepts together.

Compare and contrast:

> *The UMM CSci program is awesome.*

> *The University of Minnesota Morris Computer Science program is awesome.*

Code abbreviations

As in English, it can be useful to have abbreviations in code.

`for` is an abbreviated `while`.

`for` is built into many programming languages, but not all abbreviations are.

Macros allow the programmer to add their own abbreviations to a programming language.

```
for (int i=0;i<10;i++) {
  doStuff(i);
}
```

```
int i = 0;
while (i < 10) {
  doStuff(i);
  i++;
}
```

Outline

The outline for the rest of the talk:

- Macros and What Can Go Wrong
- Macros in the C programming language
- Macros in the Scheme programming language
- Macros in the Racket programming language
- Conclusions

The Essence of Macros

Like abbreviations, macros perform source-to-source transformations on the program.

The compiler *macro expands* source code by running all macros that appear in it. The *macro expansion* of a macro is what a macro stands for.

A Simple Example

`unless` is the same as `if`, but with a negated conditional.

`unless <test>: <body>` $\longrightarrow$ `if not <test>: <body>`

Before: `unless fileEmpty(f): readData(f)`
After: `if not fileEmpty(f): readData(f)`

In most languages, `unless` cannot be a function because functions evaluate their arguments before they are called. (Call-by-value) `unless(fileEmpty(f), readData(f))` would attempt to read from `f` even if it was empty.

What Can Go Wrong

The macro expansion is used in place of the macro. The expansion must behave correctly in that context.

Variables names may be duplicated, or code may be evaluated too many times. Both can lead to hard-to-find errors.

The first is especially bad as it only occurs when the programmer picks exactly the wrong name for a variable.

The Concept of Hygiene

A macro is *hygienic* if its macro expansion does not cause these types of errors.

The ability to write hygienic macros in a language is crucial if macro are to be a reliable part of the language.

Can be done by hand, but some languages have a *hygienic macro system*, making macros automatically be hygienic.

Macros in C

C has a unhygienic macros based on text substitution.

```
#define SQ(x) x*x
```

`SQ(2+3)` $\longrightarrow$ `2+3*2+3` $\longrightarrow$ `11`

Parentheses help:

```
#define SQ(x) ((x)*(x))
```

`SQ(2+3)` $\longrightarrow$ `((2+3)*(2+3))` $\longrightarrow$ `25`

Macros in C (Cont.)

Double evaluation is an issue:

```
#define SQ(x) ((x)*(x))
int a = 4;
```

`SQ(++a)` $\longrightarrow$ `((++a)*(++a))`

`a` will become 6.

Variables clash:

```
#define SWAP(x,y) {int tmp;tmp=x;x=y;y=tmp;}
int tmp = 5, val = 3;
```

`SWAP(tmp,val)` $\longrightarrow$ `{int tmp;tmp=tmp;tmp=val;val=tmp;}`

`SWAP` does not swap the variables.

Macros in C (Cont.)

Double evaluation is an issue:

```
#define SQ(x) ((x)*(x))
int a = 4;
```

`SQ(++a)` $\longrightarrow$ `((++a)*(++a))`

`a` will become 6.

Variables clash:

```
#define SWAP(x,y) {int tmp;tmp=x;x=y;y=tmp;}
int tmp = 5, val = 3;
```

`SWAP(tmp,val)` $\longrightarrow$ `{int tmp;tmp=tmp;tmp=val;val=tmp;}`

`SWAP` does not swap the variables.

Macros in C (Cont.)

Double evaluation is an issue:

```
#define SQ(x) ((x)*(x))
int a = 4;
SQ(++a) ⟶ ((++a)*(++a))
```

a will become 6.

Variables clash:

```
#define SWAP(x,y) {int tmp;tmp=x;x=y;y=tmp;}
int tmp = 5, val = 3;
SWAP(tmp,val) ⟶ {int tmp;tmp=tmp;tmp=val;val=tmp;}
```

SWAP does not swap the variables.

The Lisp Family of Programming Languages

Includes Common Lisp, Scheme, and Racket (a dialect of Scheme, formerly PLT/Dr Scheme). They are *functional* languages, as compared to *imperative* languages such as Java and C.

They are well-known for their macros. Scheme and Racket have hygienic macro systems, Common Lisp has facilities for manually adding hygiene.

Source code is comprised of lists. Prefix notation is used (the function is first). $2 \cdot 3 + 1$ is `(+ (* 2 3) 1)`

syntax-rules

Macros are defined in Scheme using `syntax-rules`. It defines hygienic macros using pattern matching.

We will consider a simplified subset of `syntax-rules`: `(syntax-rules (`*rule template*`)+)`.

Rules are tried in order. Once a successful match is found, the template is filled in and becomes the macro expansion.

syntax-rules Example

Suppose we wanted a `->` macro such that:

```
(-> 2 (* 3) (+ 1))
```

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

syntax-rules Example

Suppose we wanted a `->` macro such that:

`(-> 2 (* 3) (+ 1))` $\longrightarrow$ `(-> (* 2 3) (+ 1))`

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

syntax-rules Example

Suppose we wanted a `->` macro such that:
`(-> 2 (* 3) (+ 1))` $\longrightarrow$ `(-> 6 (+ 1))`

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

syntax-rules Example

Suppose we wanted a `->` macro such that:

`(-> 2 (* 3) (+ 1))` $\longrightarrow$ `(+ 6 1)`

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

syntax-rules Example

Suppose we wanted a `->` macro such that:

`(-> 2 (* 3) (+ 1))` $\longrightarrow$ `7`

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

Expanding a syntax-rules Macro

Example: `(-> 2 (* 3) (+ 1))`
Rule: `(-> form (f args ...))`
Template: `(f form args ...)`

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

Expanding a syntax-rules Macro

Example: `(-> 2 (* 3) (+ 1))`
Rule: `(-> form next-form forms ...)`
Template: `(-> (-> form next-form) forms ...)`

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

Expanding a syntax-rules Macro

Example: `(-> 2 (* 3) (+ 1))`

Rule: `(-> form next-form forms ...)`

Template: `(-> (-> form next-form) forms ...)`

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

Expanding a syntax-rules Macro

Example: `(-> 2 (* 3) (+ 1))`
Rule: `(-> form next-form forms ...)`
Template: `(-> (-> 2 next-form) forms ...)`

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

Expanding a syntax-rules Macro

Example: `(-> 2 (* 3) (+ 1))`
Rule: `(-> form next-form forms ...)`
Template: `(-> (-> 2 (* 3)) forms ...)`

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

Expanding a syntax-rules Macro

Example: `(-> 2 (* 3) (+ 1))`

Rule: `(-> form next-form forms ...)`

Template: `(-> (-> 2 (* 3)) (+ 1))`

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

Expanding a syntax-rules Macro

Example: `(-> 2 (* 3) (+ 1))`
Rule: `(-> form next-form forms ...)`
Template: `(-> (-> 2 (* 3)) (+ 1))`

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

Expanding a syntax-rules Macro

`(-> (-> 2 (* 3)) (+ 1))`

Expands to `(+ (-> 2 (* 3)) 1)`

Expands to `(+ (* 2 3) 1)`

```
(define-syntax ->
  (syntax-rules
   ((-> form (f args ...))
    (f form args ...))
   ((-> form next-form forms ...)
    (-> (-> form next-form) forms ...))))
```

Achievements And Room For Improvement

`syntax-rules` defines hygienic macros, and the macros look like what they produce.

`syntax-rules` has stood the test of almost 20 years of use.

`syntax-rules` is unable to notice a large range of syntax errors.

`syntax-rules` does not provide good feedback on bad syntax.

syntax-parse

Introduced by Northeastern researchers Ryan Culpepper and Matthias Felleisen in 2010.

Has been in Racket for about a year.

Has more advanced patterns; pattern variables are tagged with a *syntax class*, such as a list, a number, etc. Allows for user-defined syntax classes.

Automatically generates good error messages from the descriptions of syntax classes.

let As a Macro

`lambda` is the keyword in Scheme and Racket for introducing an anonymous function. `(lambda (x y) (+ x y))`

`let` is a way to introduce local variables in Scheme and Racket.

```
(let ((x 3) (y 2)) (+ x y))
```

`let` can be written in terms of `lambda`:

```
(let ((x 3) (y 2)) (+ x y))
```

$\longrightarrow$ `((lambda (x y) (+ x y)) 3 2)`

let As a Macro

When defined using `syntax-rules`

```
(let ((x 1) (x 2)) (h x))
```
lambda: duplicate argument name in: `x`
```
(let (((x y) (f 7))) (g x y))
```
reference to an identifier before its definition: `y`
```
(let (x 5) (add1 x))
```
let: bad syntax in: `(let (x 5) (add1 x))`
```
(let 17)
```
let: bad syntax in: `(let 17)`

let As a Macro

When defined using `syntax-parse`

```
(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`

Conclusions

Macros allow the programmer to extend a programming language with additional constructs.

A macro must be hygienic for it to behave correctly.

Hygienic macro systems give hygiene for free.

syntax-parse allows macros to produce good error messages.

References

W. Clinger and J. Rees. Macros that work. POPL '91. 1991

E. Kohlbecker, D. P. Friedman, M. Felleisen, B. Duba. Hygienic macro expansion. LFP '86. 1986

E. E. Kohlbecker, M. Wand. Macro-by-example. POPL '87. 1987

R. Culpepper, M. Felleisen. Fortifying macros. ICFP '10. 2010