Macros: User-Extensible Syntax

Macro = user-defined syntactic sugar

• A **macro definition** describes how to transform some new syntax into different syntax in the source language.

• A **macro system** is a language (or part of a larger language) for defining macros.

• **Macro expansion** is the process of rewriting the syntax for each **macro use**.
  – Before a program is run (or even compiled)

Example Racket Macros

Definitions:

– Expand (my-if e1 then e2 else e3) to (if e1 e2 e3)
– Expand (comment-out e1 e2) to e2

It is like we added keywords to our language

– Other keywords only keywords in uses of that macro
– Syntax error if keywords misused
– Rewriting (“expansion”) happens before execution

Uses:

  (my-if x then y else z) ; (if x y z)
  (my-if x then y then z) ; syntax error
  (comment-out (car null) #f)
Overuse

Macros sometimes get a bad wrap for being overused.

Rule of thumb:
Use macros only where functions would be awkward or impossible.

They can be useful!

Macro system design:
Tokenization

Macro systems generally work at the level of tokens not sequences of characters
– So must know how programming language tokenizes text

Example: “macro expand head to car”
– Would not rewrite (+ headt foo) to (+ cart foo)
– Would not rewrite head-door to car-door
  • But would in C where head-door is subtraction

Macro system design:
Associativity and Parenthesization

C/C++ preprocessor example:

```
#define ADD(x,y) x+y
```

ADD(1, 2/3)*4
means 1 + 2 / 3 * 4 not (1 + 2 / 3) * 4

"Solved" with emphatic parentheses by the programmer:

```
#define ADD(x,y) ((x)+(y))
```

Racket S-expression syntax trivially avoids this problem!

Macro system design:
Local bindings and shadowing

Suppose macros also apply to variable bindings.
Then: “macro expand head to car”

```
(let ([head 0][car 1]) head) ; 0
(let* ([head 0][car 1]) head) ; 0
```

Would become:

```
(let ([car 0][car 1]) car) ; error
(let* ([car 0][car 1]) car) ; 1
```

C/C++ "safety by convention": all-caps macros and non-all-caps everything else

Racket does not work this way – it gets scope “right”!
Example Racket macro definitions

Two simple macros

```racket
(define-syntax my-if
  (syntax-rules (then else)
    [(my-if e1 then e2 else e3) (if e1 e2 e3)]))
```

If a syntactic form matches, do the corresponding expansion
- In these examples, list of possible use forms has length 1
- Else syntax error

```racket
(define-syntax comment-out
  (syntax-rules ()
    [(comment-out ignore instead)])

```

Macro style

Equivalent functions that double their argument:

```racket
(define (dbl x) (+ x x))
(define (dbl x) (* 2 x))
```

Bad style as a macro:

```racket
(define-syntax dbl
  (syntax-rules()
    [(dbl x) (* 2 x)]))
```

1. If a function works fine, don't use a macro.
2. Avoid surprising reevaluation:
   ```racket
   (dbl (begin (print "hi") 42))
   ```

Hygiene: avoid accidental shadowing

In C/C++, defining local variables inside macros is unwise
- When needed done with hacks like __strange_name34

Example:

Macro:
```racket
(define-syntax dbl
  (syntax-rules()
    [(dbl x) (let ([y 1]) (* 2 x y))]))
```

Use:
```racket
(let ([y 7]) (dbl y))
```

Naïve expansion:
```racket
(let ([y 7]) (let ([y 1]) (* 2 y y)))
```

Racket hygienic macros avoid this problem.
Hygiene: avoid accidental shadowing

Example:
Macro:
```
(define-syntax dbl
  (syntax-rules ()
    [(dbl x) (* 2 x)]))
```

Use:
```
(let ([* +]) (dbl 42))
```

Naïve expansion:
```
(let ([* +]) (* 2 42))
```

Racket hygienic macros avoid this problem.

Maintaining macro hygiene

A hygienic macro system:
1. Secretly renames local variables in macros with fresh names
2. Looks up variables used in macros where the macro is defined

Neither of these rules are followed by the “naïve expansion” most macro systems use
   – Without hygiene, macros are much more brittle (non-modular)

On rare occasions, hygiene is not what you want
   – Racket has somewhat complicated support for that

Sound familiar? Analogous to __________ vs. __________.

More examples in code: for loop, less parensy lets, let* as sugar.