Macros

In this section we'll discuss the powerful macro system that Racket offers. We will start by looking at preprocessor macros in C.

Macros in C

You are probably already familiar with preprocessor macros in C[1]. They are defined using the preprocessor directive `#define` and essentially the preprocessor goes through the rest of the file and replaces any occurence of the defined symbol with the expansion. Here's a simple example:

```
#define BUFFER_SIZE 1024

foo = (char *) malloc (BUFFER_SIZE);
```

This is exactly the same as if you had literally written:

```
foo = (char *) malloc (1024);
```

The first thing that is troublesome about the preprocessor is that it allows you to even overwrite keywords:

```
#include <stdio.h>
#define true 0

int main () {
   if (true) { printf("true!\n"); }
   else { printf("false!\n"); }
   return 0;
}
```

The expansion is purely syntactic, and it includes everything that follows the line. So for example the following is valid and prints 7:

```
#define A 2 +
int main() { printf("%d\n", A A 3); return 0; }
```

As you can imagine, there are many ways to abuse the system.

You can also create macros that behave a bit more like functions:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
z = min(a + 28, *p);      /* becomes: z = ((a + 28) < (*p) ? (a + 28) : (*p))      */
```

The parentheses might sound overkill, but not having them there could cause trouble and result in unpredictable behavior. For instance:

```
#define mult(a, b) a * b
z1 = mult(3 + 2, 5)       /* becomes: 3 + 2 * 5 = 13, not (3 + 2) * 5 = 25 */
```

Here's some other crazy ones, that you can try at home, borrowed from this gist[2]:

[1] https://gcc.gnu.org/onlinedocs/cpp/Macros.html
[2] https://gist.github.com/aras-p/6224951

```
#define for for(int z=0;z<2;++z)for
#define struct union
#define if while
#define else
#define break
#define if(x)
#define double float
#define delete
#define volatile // this one is cool
#define true ((__LINE__&15)!=15)
#define true ((rand()&15)!=15)
#define if(x) if ((x) && (rand() < RAND_MAX * 0.99))
#define sizeof(x) (sizeof(x)-1)
#define strcpy(a,b) memmove(a,b,strlen(b)+2)
#define continue break
```

Racket Macros

Racket offers a very powerful and safer macro system, that includes what are called **hygienic macros**; we will discuss the concept shortly. Let us start with some examples.

First, here is an example that allows us to write the words then and else as part of an if. This macro will convert (my-if e1 then e2 else e3) to the Racket-correct (if e1 e2 e3).

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

We define a macro with define-syntax. The first term after it is the name of the macro. We then have a syntax-rules call that specifies a series of "keywords" that appear in the macro that should not be treated as variables, in this case those are then and else. The remaining arguments to syntax-rules is a series of bracketed pairs of expressions, where the first part of the pair is the form of my-if and the second is what it should be replaced by. We can create a single macro that can accept multiple forms of argument and behaves differently for each one.

A key thing in the above is that the expressions e1, e2 and e3 are not being evaluated when my-if is encountered; the conversion to if happens without the evaluation of the expressions, which is good because we wouldn't want e2 and e3 to always be evaluated. In this way macros are very different from functions. For example here is a macro for "or":

```
(define-syntax or
  (syntax-rules ()
    [(or e1 e2)
     (if e1 e1 e2)]))  ;; Assumes "or" simply returns the first value that is not #f
```

A particularly nice use of macros is to build a "better" delay function, one that doesn't require us to provide a thunk. We can simply provide the function we want to delay. Again, we use the fact that, unlike functions, macros don't evaluate they arguments.

```
(define-syntax delay2
  (syntax-rules ()
    [(delay e)
     (mcons #f (lambda () e))]))
```

One thing to watch out for is exactly the fact that macros don't actually evaluate their expressions. For instance consider the following macro for squaring a number:

```
(define-syntax sq
  (syntax-rules ()
    [(sq e)
     (* e e)]))

(sq (begin (print "You'll see this twice!") 4))
```

The macro expansion simply replaces both occurences of `e` the argument passed to `sq`, without evaluating it first. Then when the multiplication occurs both terms will be evaluated, resulting in a double evaluation of the `begin` block. One way around it is to use a let expression as part of the substitution:

```
(define-syntax sq2
  (syntax-rules ()
    [(sq e)
     (let ([v e])
       (* v v))]))

(sq2 (begin (print "You'll see this only once now!") 4))
```

Hygienic Macros

Creating new variable bindings within a macro can create some interesting naming problems. For example consider the following:

```
(define-syntax foo
  (syntax-rules ()
    [(foo e)
     (let ([v (+ e 1)])
       (+ v e))]))

(let ([v 5])
  (foo v))
```

Then according to naive macro-expansion rules the last two lines would become:

```
(let ([v 5])
  (let ([v (+ v 1)])
    (+ v v)))
```

The result of this would be `12`, as the `v` in the inner `let` would get the value `6` and the addition would use that `v` both times. That is clearly not what we intended. We would want the second `v` in `(+ v v)` to be the one inherited from the outer `let`. A naive macro-expander would end up making a hard to spot mistake.

The problem of course is this: identifiers bound inside the macro may happen to also be contained in the expressions that are passed as arguments, and the fact that these expressions are substituted as is rather than evaluated would cause name conflicts.

A *hygienic macro system*, like the one Racket has, automatically changes internally the names of bound identifiers to avoid these conflicts. So in the example above, it would create instead:

```
(let ([v 5])
  (let ([v2 (+ v 1)])
    (+ v2 v)))
```

Assuming `v2` has not been bound previously. This would produce the correct result of `11`.

Finally, here's a way to implement `if` using `cond`:

```
(define-syntax if
  (syntax-rules ()
    [(if e1 e2 e3)
     (cond [e1 e2]
           [#t e3])]))
```

And we will also do the converse, converting a `cond` into a sequence of `if`s. This is trickier, and requires providing multiple alternative rules, and using the `...` argument which allows us to represent "the rest of the entries":

```
(define-syntax cond
  (syntax-rules (else)
    [(cond)
     (error "cond: no matching case")]
    [(cond [else e])
     e]
    [(cond [e1 e2] rest ...)
     (if e1 e2 (cond rest ...))]))
```

Links to learn more

Some more links to learn about macros:

- Fear of Macros[3]
- Racket's Macros docs[4]

[3] http://www.greghendershott.com/fear-of-macros/
[4] https://docs.racket-lang.org/guide/macros.html