Macros

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

Motivation

- Even in the best software designs, it's hard to avoid repetitive patterns.
- What if our language let us extend its syntax to account for these patterns?

Exercise for Home

Find a piece of code you wrote (in any language) which repeats a syntax pattern you couldn't avoid by writing a function, class, etc.

What do I mean by "extend syntax"?

We can implement most all of the functionality we need in Python using functions. But can we implement something like Racket's `let` in Python?

```
let (x = 10,
     y = 20) in:
   print(x, y)
```

(Python does not support above)

How about C Macros?

The C Preprocessor lets us do simple text substitutions:

```
#define FOREVER for (;;)

main () {
    FOREVER {
        printf("Hello, World!\n");
    }
}
```

(they can get a little more complicated than that...)

But what happens when we want to do more complex things?
Like manipulate the body of that "FOREVER loop"?

C Macros

At some point, textual source manipulation cannot serve the purpose we need anymore. Let this source from MicroPython serve as an example:

```
STATIC mp_obj_t machine_spi_init(...) {
    ...
}
STATIC MP_DEFINE_CONST_FUN_OBJ_KW(machine_spi_init_obj, 1, mac

STATIC mp_obj_t machine_spi_deinit(...) {
    ...
}
STATIC MP_DEFINE_CONST_FUN_OBJ_1(machine_spi_deinit_obj, machi

STATIC mp_obj_t mp_machine_spi_read(...) {
    ...
}
```

Lisp Macros

Hopefully it's become apparent that **symbolic computation** is the right tool for the job when it comes to macros.

Lisp Macros:

- Compile time
- Syntax $\rightarrow$ Syntax

Lisp Functions:

- Run time
- Data $\rightarrow$ Data

- Lisp dialects usually make the run time available during the compile time, so the normal language can be used to write macros.

Macros as Syntax Transformers

```
YOU ARE INSIDE A ROOM.
THERE ARE KEYS ON THE GROUND.
THERE IS A SHINY BRASS LAMP NEARBY.

IF YOU GO THE WRONG WAY, YOU WILL BECOME
HOPELESSLY LOST AND CONFUSED.

> pick up the keys

YOU HAVE A SYNTAX TRANSFORMER
```

Old-School Lisp Macros

Early Lisp macro systems operated on the simple contract of functions which take syntax, manipulate it, and returns a list containing the new syntax:

```
(defmacro repeat-forever (&rest body)
  `(prog ()
     a   ,@body
         (go a)))

;; we can then use the macro like this:
(repeat-forever
  (format t "HELLO WORLD~%"))
```

Old-School: More Examples

"let" as a macro:

```
(defmacro let (bindings &rest body)
  `((lambda ,(mapcar #'car bindings)
      ,@body)
    ,@(mapcar #'cadr bindings)))

;; we can then use let like this:
(let ((a 10)
      (b 20))
  (format t "~A ~A~%" a b))
```

Old-School: Another Example

Suppose we wanted to define a syntax like this:

```
(numeric-case num
  negative
  zero
  positive)
```

We could write a macro like this:

```
(defmacro numeric-case (num negative zero positive)
  `(let ((result ,num))
     (cond
       ((< result 0) ,negative)
       ((= result 0) ,zero)
       (t ,positive))))
```

What could possibly go wrong?

Fixing numeric-case with gensym

gensym is here to save us when we need really obscure symbol names:

```
(defmacro numeric-case (num negative zero positive)
  (let ((sym (gensym)))
    `(let ((,sym ,num))
       (cond
         ((< ,sym 0) ,negative)
         ((= ,sym 0) ,zero)
         (t ,positive)))))
```

More Macro Issues

- What happens if the programmer redefined one of the functions we used (e.g., < or =) in the previous example?

Unhygienic Macros

Modern Lisp dialects typically provide what is called **hygienic macros**: macro systems which eliminate the issues we discovered with old-school Lisp macros (to varying degrees)

Racket's Hygienic Macros

- `define-syntax` defines compile-time syntax: a function that takes a "syntax" and returns a "syntax".
- Typical syntax operations provide a convenient way to manipulate the syntax in a hygenic manner.
- You can also go unhygienic: `syntax->datum` converts syntax to lists, symbols, etc., and `datum->syntax` goes back.

What is a "syntax"?

Syntax literals can be written using `#'`[1]:

```
> #'(if (> 0 x) y z)
#<syntax:readline-input:1:2 (if (> 0 x) y z)>
> (define stx #'(if (> 0 x) y z))
```

We can convert this to a list if we wish:

```
> (syntax->datum stx)
'(if (> 0 x) y z)
```

And back:

```
> (datum->syntax stx (syntax->datum stx))
#<syntax (if (> 0 x) y z)>
```

If you didn't have access to the original syntax object, you could pass `#f` as the first argument to `datum->syntax`.

[1] Note this is completely different from the function-namespace thing in

Going Unhygienic

We could write our `let` macro without considerations for hygiene:

```
(define-syntax (my-let stx)
  (datum->syntax
    stx
    (let ([stx-list (syntax->datum stx)])
      `((lambda ,(map car (cadr stx-list))
          ,@(cddr stx-list))
        ,@(map cadr (cadr stx-list))))))
```

A little bit yucky, but it worked.

Doing Things Hygienic

syntax-case acts like match but for syntax objects:

```
(define-syntax (my-let stx)
  (syntax-case stx ()
    [(_ ([name expr] ...) body ...)
     #'((lambda (name ...)
          body ...)
        expr ...)]))
```

define-syntax-rule Shorthand

`define-syntax-rule` is a shorthand for a `define-syntax` with a `syntax-case` of a single rule inside.

```
(define-syntax-rule (my-let ([name expr] ...) body ...)
  ((lambda (name ...)
     body ...)
   expr ...))
```

Application of Macros: Anaphora

In natural language, anaphora is a reference to a previously defined noun:

Susan dropped *the plate.* *It* *shattered loudly!*

the plate.: *referent*; *It*: *anaphor*

Lisp programmers call a similar technique the same name:

```
(printf "~a~%"
        (aif (member 10 lst)
          it
          "10 not in the list"))
```

Available in a Racket Package

The "anaphoric" package provides `aif`, `awhen`, `acond`, and `aand`.

Anaphoric If

Example from "Fear of Macros", which you will read for the LGA this weekend.

```
(require racket/stxparam)

(define-syntax-parameter it
  (lambda (stx)
    (raise-syntax-error (syntax-e stx) "outside of anaphora")))

(define-syntax-rule (aif predicate consequent alternative)
  (let ([result predicate])
    (if result
        (syntax-parameterize ([it (make-rename-transformer #'r
          consequent)
        alternative)))
```