Lecture #29: Defining Syntax—Macros

- In effect, function and class definitions extend the Python language by adding new commands and data types.
- However, these are highly constrained extensions.
- For example, there is no way to define

```
def swap(x, y):
    """Swap the values of variables X and Y."""
    ????
```

  because Python uses call-by-value.
- Likewise, there is generally no way to define a new control construct.
- Indeed, language extension can be dangerous; it's easy to get wrong and can make programs less easy to read or understand.

Macros

- A *macro* is a programming-language construct that allows one to define, in effect, a function that *generates program text* that is substituted for "calls" on the macro function.
- For example (making up some new Python syntax):

```
defmacro swap(x, y):
    x, y = y, x
```

- A call on this macro, such as

```
swap(a[i], a[k])
```

  would be *expanded* into

```
a[i], a[k] = a[k], a[i]
```

  which is what actually gets executed.

Simple Macro Features

- The (imaginary) `defmacro` construct is essentially the same as the macro facilities of *C* and *C++*.
- In those languages, the definition

```
#define BLUE 3
```

  simply causes '`3`' to be substituted for the identifier '`BLUE`' wherever it appears.

- And definitions such as

```
#define doList(Var, List) \
   for (LinkedList* Var = List; Var != NULL; Var = Var->next)
```

  expands

```
doList(A, myList)
```

  into

```
   for (LinkedList* A = myList; A != NULL; A = A->next)
```

C Macro Implementation

- These substitutions are performed in C and C++ by a *preprocessor* program before standard compilation takes place.
- The preprocessor performs substitutions and deletes all the macro-definition statements (as well as C/C++ comments).
- These macros do not observe scope rules; the macro preprocessor actually knows almost nothing about C.
- In fact, one can use the C preprocessor as a separate program on any kind of textual input data.

Bells and Whistles

- Aside from simple substitution of macro parameters, C/C++ macros provide very little in the way of text processing...
- ...aside from "stringification":

```
#define defsym(x)  x = #x
defsym(y)    expands into     y = "y"
```

- ...and token concatenation:

```
#define doArray(var, A, low, high) \
    for (int var ## _index = low; var ## _index < high; \
         var ## _index += 1) { \
       int var = (A)[var ## _index];
#define endDo  }
```

This example allows one to write things like

```
doArray(p, anArray, 0, N)
    printf("Item %d is %d.\n", p_index, p);
endDo
```

Conditional Compilation

- The C macro preprocessor also provides statements like this:

```
#if defined(NDEBUG)
    #define assert(Test, Message)
#else
    #define assert(Test, Message) \
        if (!(Test)) { \
            fprintf(stderr, "%s\n", Message); \
            abort(1); \
        }
#endif
```

- This example says that if a macro named `NDEBUG` is defined, we define a macro named `assert` to do nothing (it expands to nothing), and otherwise it expands to a statement that tests whether an expression `Test` is true, and exits with an error message if it isn't.
- Thus, when `NDEBUG` is defined, all assertions in the program are "turned off" and consume no execution time.
- This facility is called *conditional compilation*. Everything here happens *before* any execution of the program.

Scheme Macros

- The Lisp family has its own version of macro processing, one that is far more powerful than that of C.
- Full Scheme provides a powerful (but rather tricky) way to create new special forms: `define-syntax`.
- One of the extensions of our project is a simpler, more traditional form of this: `define-macro`.
- Macros are like functions, but
  - Do not evaluate their arguments (this is what makes them special forms).
  - Automatically treat the returned value as a Scheme expression and execute it.
- Thus, macros "write" programs that then get executed.

First: Quasiquote

- Writing programs that write programs entails constructing Scheme expressions that often contain substantial constant parts (that one would like to write as ordinary Scheme lists) with pieces that are computed and differ from one expansion to another.
- For this purpose, it is convenient to have a minilanguage that allows one to write expressions that resemble the expressions they produce.
- With quasiquote, I can write

```
(list 'a 'b (+ 2 3) 'd)        ;; (a b 5 d)
```

as

```
`(a b ,(+ 2 3) d)              ;; That's a backquote in front
```

- That is, everything preceded by a comma is replaced by its value.
- Additionally, in place of

```
(define values (list (+ 2 3) (- 2 1)))
(append '(a b) values '(d))    ;; which produces (a b 5 1 d)
```

I can write

```
`(a b ,@values d)      or    `(a b ,@(list (+ 2 3) (- 2 1)) d)
```

- Expressions preceded by ',@' are evaluated and their (list) values spliced in.

Very Simple Example

- Suppose we'd like to be able to write

```
>>> (define x 3)
x
>>> (unless (list? x) (displayln x))
3
```

  instead of

```
(if (not (list? x)) (displayln x))
```

- We can't define `unless` as a function, because we don't always want to evaluate `(displayln x)`.
- So instead, we write

```
(define-macro (unless cond body)
   `(if (not ,cond) ,body))
```

- If we evaluate

```
(unless (list? x) (displayln x))
```

  the `unless` macro first computes

```
(if (not (list? x)) (displayln x))
```

  which, rather than being returned, is first evaluated to print 3.

Macro Example

- Suppose we want to define something like Python's list comprehensions for Scheme.
- For example, I'd like to be able to write

```
>>> (define L '(1 2 3 4 5))
>>> (for-list x (* x x) L)   ;; Like [ x*x for x in L ]
(1 4 9 16 25)
```

- Here, we don't want to evaluate `x` or `(* x x)` immediately. Otherwise, it's a shorthand for `map`.
- We can write it as a macro:

```
(define-macro (for-list var expr lst)
    `(map (lambda (,var) ,expr) ,lst))
```

- So each time we evaluate

```
(for-list x (* x x) '(1 2 3 4 5))
```

  it first computes the list

```
(map (lambda (x) (* x x)) '(1 2 3 4 5))
```

  and then evaluates that to get the result.

Macro Example (II)

- Along the same lines, we might do some sort of traditional counting loop: comprehensions for Scheme.
- I'd like to be able to write things like

```
>>> (for-range x 1 5 (* x x))  ;; Like [ x*x for x in range(1, 6) ]
(1 4 9 16 25)
```

```
(define-macro (for-range control-var low high body)
    `(let (($low$ ,low))
       (define ($loop$ $so-far$ ,control-var)
           (if (< ,control-var $low$) $so-far$
              ($loop$ (cons ,body $so-far$) (- ,control-var 1))))
       ($loop$ '() ,high)))
```

- So `(for-range x 1 5 (* x x))` first expands into

```
(let (($low$ 1))
    (define ($loop$ $so-far$ x)
        (if (< x $low$) $so-far$
           ($loop$ (cons (* x x) $so-far$) (- x 1))))
    ($loop$ '() 5))
```

which is then evaluated to produce the desired list.

Macro Example (III)

- We could also imagine combining these two, so as to be able to write either of

```
>>> (for (x (list 1 2 3 4 5)) (* x x))
(1 4 9 16 25)
>>> (for (x 1 5) (* x x))
(1 4 9 16 25)
```

- That is, the macro takes its cue from whether the list specifier has two arguments (the second assumed to be a list) or three (assumed to be an integer range):

```
(define-macro (for list-spec body)
    (let ((control-var (car list-spec))
          (opnds (cdr list-spec)))
       (if (= (length opnds) 1)
        `(for-list ,control-var ,body ,(car opnds))
        `(for-range ,control-var ,(car opnds) ,(car (cdr opnds)) ,body))))
```

- This is an example of conditional compilation in Scheme. The `for` macro expands either into `for-map` or `for-range`, depending on the number of elements in the `list-spec` parameter.

Name Clashes

- The unnecessary use of macros has long been discouraged, because they introduce some serious issues.
- Consider our list comprehension example again:

```
(define-macro (for-range control-var low high body)
   `(let (($low$ ,low))
      (define ($loop$ $so-far$ ,control-var)
         (if (< ,control-var $low$) $so-far$
            ($loop$ (cons ,body $so-far$) (- ,control-var 1))))
      ($loop$ '() ,high)))
```

- The identifier `$loop$` is intended to be local to the macro. I gave it a funny name to make it unlikely that it will conflict with any names the programmer has used in `body`.
- But there's no guarantee that I've succeeded in preventing a clash.
- A problematic example:

```
>>> (define $low$ 15)
>>> (for-range x 1 5 (* $low$ x))
>>> (1 2 3 4 5)    ;; WRONG!!  Should be (15 30 45 60 75)
```

Name Clashes (II)

- One solution: some Lisp dialects supply a builtin function that generates new symbols that are guaranteed to differ from all other symbols.

```
(define-macro (for-range control-var low high body)
    (let ((low-name (gensym))
          (so-far-name (gensym))
          (loop-name (gensym)))
       `(let ((,low-name ,low))
          (define (,loop-name ,so-far-name ,control-var)
              (if (< ,control-var ,low-name) ,so-far-name
                 (,loop-name (cons ,body ,so-far-name)
                             (- ,control-var 1))))
          (,loop-name '() ,high))))
```

- In the above, I've replaced the fixed symbols `$low$`, `$so-far$`, and `$loop$` with freshly generated symbols for each use of the macro.