Extra Lecture #7: Defining Syntax

- 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, the is no way to define

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

  because Python used 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.
- 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)        ;; which produces (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)
```

- That is everything preceded by ',@' is evaluated and its (list) value spliced in.

Macro Example

- We may define a new looping construct:

```
(define-macro (while cond stmt)
    `(begin (define ($loop$) (if ,cond (begin ,stmt ($loop$))))
            ($loop$)))
```

- So `(while (> x y) (set!  x (f y)))` first yields

```
(begin (define ($loop$)
             (if (> x y) (begin (set! x (f y)) ($loop$))))
       ($loop$))
```

- And then this is executed.

A Macro for Streams

- Syntax extension allows us to define a convenient kind of stream in Scheme.
- As we did in Python, a stream in Scheme will consist of a head, and either a function to compute the tail or the tail itself.

```
(define-macro (cons-stream head tail)
    `(cons ,head (lambda () ,tail)))
```

- We'll need a special cdr function that calls the tail computation (if it is a function).

```
(define (cdr-stream str)
    (if (procedure? (cdr str))
        ; Compute and memoize tail
        (set-cdr! str ((cdr str))))
    (cdr str))
```

- Actually, these are built into our (fully extended) project.

Streams in Scheme

```
;; The stream of all 1's
(define ones (cons-stream 1 ones))
(car ones) ===> 1
(car (cdr-stream ones)) ===> 1

(define (add-streams a b) ; Infinite streams, that is
    (cons-stream (+ (car a) (car b))
                 (add-streams (cdr-stream a) (cdr-stream b))))

;; The stream 1 2 3 ...
(define nums ?)

;; The Fibonacci sequence
(define fib (cons-stream 1
              (cons-stream 1
                 ?)))
```

Streams in Scheme

```
;; The stream of all 1's
(define ones (cons-stream 1 ones))
(car ones) ===> 1
(car (cdr-stream ones)) ===> 1

(define (add-streams a b) ; Infinite streams, that is
    (cons-stream (+ (car a) (car b))
                 (add-streams (cdr-stream a) (cdr-stream b))))

;; The stream 1 2 3 ...
(define nums (cons-stream 1 (add-streams ones nums)))

;; The Fibonacci sequence
(define fib (cons-stream 1
               (cons-stream 1
                  ?)))
```

Streams in Scheme

```scheme
;; The stream of all 1's
(define ones (cons-stream 1 ones))
(car ones) ===> 1
(car (cdr-stream ones)) ===> 1

(define (add-streams a b) ; Infinite streams, that is
    (cons-stream (+ (car a) (car b))
                 (add-streams (cdr-stream a) (cdr-stream b))))

;; The stream 1 2 3 ...
(define nums (cons-stream 1 (add-streams ones nums)))

;; The Fibonacci sequence
(define fib (cons-stream 1
                (cons-stream 1
                   (add-streams fib (cdr-stream fib)))))
```

Name Clashes

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

```
(define-macro (while cond stmt)
    `(begin (define ($loop$) (if ,cond (begin ,stmt ($loop$))))
            ($loop$)))
```

- 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.
- But there's no guarantee that I've succeeded in preventing a name clash.
- One solution: some Lisp dialects supply a builtin function that generates new symbols that are guaranteed to differ from all other symbols.

```
(define-macro (while cond stmt)
    (define loop-sym (gensym))
    `(begin (define (,loop-sym) (if ,cond (begin ,stmt (,loop-sym))))
            (,loop-sym)))
```

Real Scheme Approach

- Real Scheme allows a general syntax-definition construct that creates local variables as needed (among other things).

```
(define-syntax while
  (syntax-rules ()
    ((_ pred b1 ...)
     (let loop () (when pred b1 ... (loop))))))
```