

# Syntactic Abstraction in Scheme

- Programming languages should be flexible, easy to use, easy to read.
- *Syntactic sugar* are methods that make a language easier to use that do not increase its fundamental capacity.

Example:

```
(let ((var1 exp1)  
      ...  
      (varn expn))  
  body )
```

is equivalent to

```
((lambda (var1 ... varn) body)  
  exp1 ... expn)
```

## More on let

Let evaluates its arguments in parallel: The scope of  $var_1, \dots, var_n$  is restricted to *body*. In

```
(define x 10)
(let ((x 2)
      (y (+ 3 x))
      (* x y)),
```

what value does let return? What about in

```
(define x 10)
(let ((x 2))
  (let ((y (+ 3 x)))
    (* x y)))?
```

Why aren't these equivalent?

## More on let

let can also be used to define local functions. What value does

```
(let ((add2 (lambda (x) (+ x 2)))
      (x 4))
      (add2 x))
```

return?

What about

```
(let ((list-sum (lambda (lon)
                  (if (null? lon)
                      0
                      (+ (car lon)
                         (list-sum (cdr lon)))))))
      (odds '(1 3 5 7)))
(list-sum odds))
```

letrec defines local recursive functions

```
(letrec ((list-sum (lambda (lon)
                    (if (null? lon)
                        0
                        (+ (car lon)
                          (list-sum (cdr lon)))))))
  (odds '(1 3 5 7)))
(list-sum odds))
```

returns 16.

## More on letrec

In

```
(letrec ((var1 exp1)  
        ...  
        (varn expn))  
  body )
```

the scope of  $var_1, \dots, var_n$  is the entire letrec expression; however these variables may not be referenced during the evaluation of  $exp_1, \dots, exp_n$ . For example

```
(letrec ((x 3)  
        (y (lambda() (+ x 1))))  
  (y))
```

is legal, but

```
(letrec ((x 3)  
        (y (+ x 1)))  
  y)
```

is not.

## Logical connectives (special forms and & or)

Expression

`(and test1 test2 ... testn)`

is equivalent to

`(if test1  
 (and test2 ... testn)  
 #f)`

`(or test1 test2 ... testn)`

is equivalent to

```
(let ((*value* test1))
  (if *value*
      *value*
      (or test2 ... testn)))
```

(assuming that `*value*` does not appear as a free variable in `test2, ... testn`).

## Conditionals

```
(cond
  (test1 consequent1)
  ...
  (testn consequentn)
  (else alternative))
```

is equivalent to

```
(if test1
    consequent1
    ...
    (if testn
        consequentn
        alternative) ...)
```

## Case

```
(case key
  (key-list1 consequent1)
  ...
  (key-listn consequentn)
  (else alternative))
```

is equivalent to

```
(let ((*key* key))
  (cond
    ((memv *key* 'key-list1) consequent1)
    ...
    ((memv *key* 'key-listn) consequentn)
    (else alternative)))
```

(assuming that *\*key\** does not appear as a free variable in the consequent or alternative expressions).

## Records

In C it is easy to create new data structures:

```
struct tag {  
    type1 field1;  
    ...  
    typen fieldn;  
};
```

Powerful features can be added to Scheme:

- (define-record *name* (*field*<sub>1</sub> ... *field*<sub>*n*</sub>))  
creates a new datatype of type *name*, that contains the specified fields; plus it creates three different sets of procedures: *make-name*, takes *n* arguments, and returns a new record; *name?* is an appropriate predicate for type *name*; and *name*->*field*<sub>*i*</sub> is an accessor function for reading and modifying field values.
- variant-case acts like a case statement, where the keyvalue is its type.