

Exceptions are raised with a raise statement.

`raise <expr>`

<expr> must evaluate to a subclass of BaseException or an instance of one.

```
try:
  <try suite>
except <exception class> as <name>:
  <except suite>
```

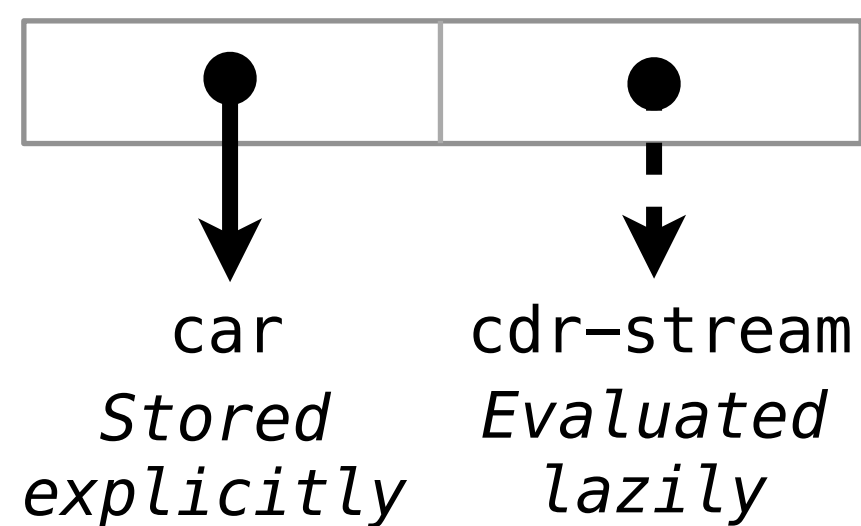
The <try suite> is executed first. If, during the course of executing the <try suite>, an exception is raised that is not handled otherwise, and

```
>>> try:
      x = 1/0
      except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0
handling a <class 'ZeroDivisionError'>
>>> x
0
```

If the class of the exception inherits from <exception class>, then The <except suite> is executed, with <name> bound to the exception.

```
(car (cons 1 2)) -> 1
(cdr (cons 1 2)) -> 2
(car (cons 1 (/ 1 0))) -> ERROR
(cdr (cons 1 (/ 1 0))) -> ERROR
```

A stream is a Scheme pair, but the cdr is evaluated lazily

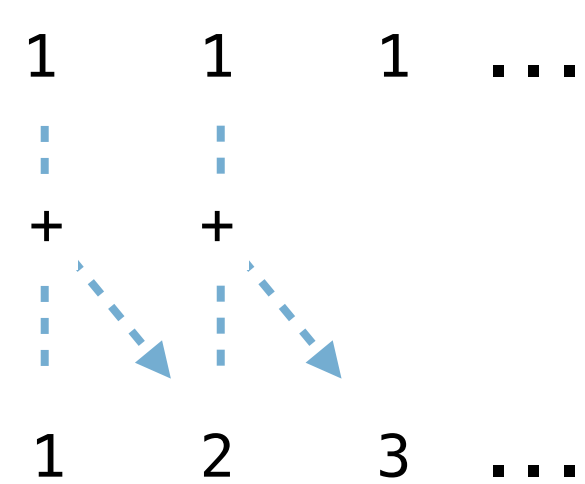


```
(car (cons-stream 1 2)) -> 1
(cdr-stream (cons-stream 1 2)) -> 2
(car (cons-stream 1 (/ 1 0))) -> 1
(cdr-stream (cons-stream 1 (/ 1 0))) -> ERROR
```

```
(define (range-stream a b)
  (if (>= a b)
      nil
      (cons-stream a (range-stream (+ a 1) b))))
(define lots (range-stream 1 10000000000000000000))
scm> (car lots)
1
scm> (car (cdr-stream lots))
2
scm> (car (cdr-stream (cdr-stream lots)))
3
```



```
(define ones (cons-stream 1 ones))
(define (add-streams s t)
  (cons-stream (+ (car s) (car t))
               (add-streams (cdr-stream s)
                             (cdr-stream t))))
```



```
(define ints (cons-stream 1 (add-streams ones ints)))
```

```
(define (map-stream f s)
  (if (null? s)
      nil
      (cons-stream (f (car s))
                   (map-stream f (cdr-stream s)))))
```

```
(define (prefix s k)
  (if (= k 0)
      nil
      (cons (car s) (prefix (cdr-stream s) (- k 1)))))
```

```
(define (filter-stream f s)
  (if (null? s)
      nil
      (if (f (car s))
          (cons-stream (car s)
                       (filter-stream f (cdr-stream s)))
          (filter-stream f (cdr-stream s)))))
```

The built-in Scheme list data structure can represent combinations

```
scm> (list 'quotient 10 2)      scm> (eval (list 'quotient 10 2))
(quotient 10 2)              5
```

A macro is an operation performed on source code before evaluation

```
(define-macro (twice expr)
  (list 'begin expr expr))
> (twice (print 2))
2
2
(begin (print 2) (print 2))
```

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- Call the macro procedure on the operand expressions
- Evaluate the expression returned from the macro procedure

```
scm> (map (lambda (x) (* x x)) '(2 3)) scm> (for x '(2 3) (* x x))
(4 9)                                  (4 9)
(define-macro (for sym vals expr) OR (define-macro (for sym vals expr)
  (list 'map (list 'lambda (sym) (map (lambda (,sym) ,expr) ,vals))
        (list sym)
        expr) vals))
```

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter should support an unbounded number of active tail calls.

A tail call is a call expression in a tail context, which are:

- The last body expression in a lambda expression
- Expressions 2 & 3 (consequent & alternative) in a tail context if
- All non-predicate sub-expressions in a tail context cond
- The last sub-expression in a tail context and, or, begin, or let

```
(define (factorial n k)
  (if (= n 0) k
      (factorial (- n 1) (* k n))))
(define (length s)
  (if (null? s) 0
      (+ 1 (length (cdr s)))))
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (length-iter (cdr s) (+ 1 n))))
  (length-iter s 0))
```

Not a tail call

Recursive call is a tail call

```
(define size 5) ;=> size
(* 2 size) ;=> 10
(if (> size 0) size (- size)) ;=> 5
(cond ((> size 0) size) ((= size 0) 0) (else (- size))) ;=> 5
((lambda (x y) (+ x y size)) size (+ 1 2)) ;=> 13
(let ((a size) (b (+ 1 2))) (* 2 a b)) ;=> 30
(map (lambda (x) (+ x size)) (quote (2 3 4))) ;=> (7 8 9)
(filter odd? (quote (2 3 4))) ;=> (3)
(list (cons 1 2) size 'size) ;=> ((1 . 2) 5 size)
(list (equal? 1 2) (null? nil) (= 3 4) (eq? 5 5)) ;=> (#f #t #f #t)
(list (or #f #t) (or) (or 1 2)) ;=> (#t #f 1)
(list (and #f #t) (and) (and 1 2)) ;=> (#f #t 2)
(append '(1 2) '(3 4)) ;=> (1 2 3 4)
(not (> 1 2)) ;=> #t
(begin (define x (+ size 1)) (* x 2)) ;=> 12
`(+ size (- ,size) ,(* 3 4)) ;=> (+ size (- 5) 12)
(force (delay (+ 1 2))) ;=> 3
```

```
;; Return a copy of s reversed.
(define (reverse s)
  (define (iter s r)
    (if (null? s) r
        (iter (cdr s)
              (cons (car s) r))))
  (iter s nil))
;; Apply fn to each element of s.
(define (map fn s)
  (define (map-reverse s m)
    (if (null? s) m
        (map-reverse (cdr s)
                      (cons (fn (car s)) m))))
  (reverse (map-reverse s nil)))
```

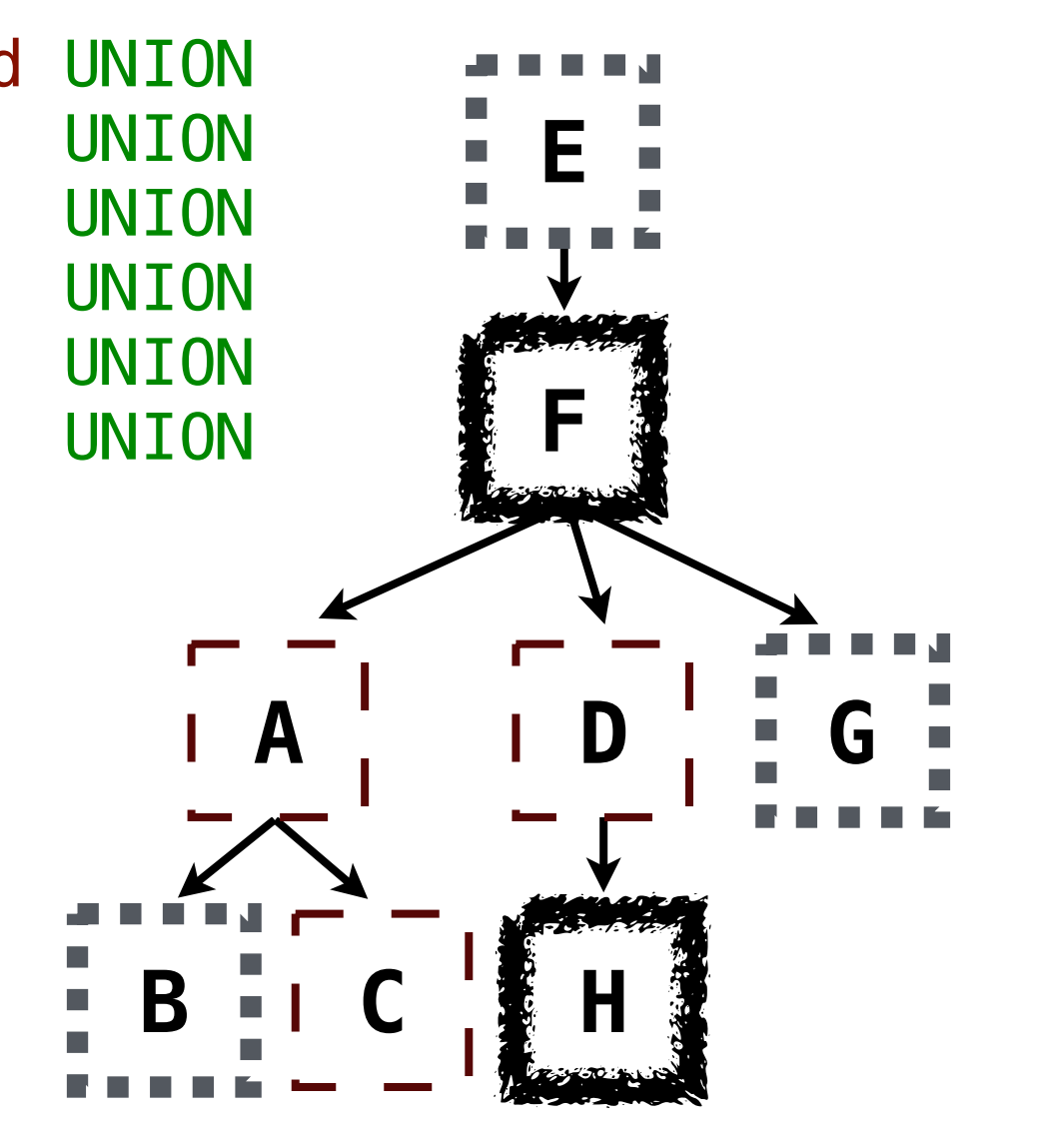
A table has columns and rows

Latitude	Longitude	Name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

A column has a name and a type

A row has a value for each column

```
SELECT [expression] AS [name], [expression] AS [name], ...;
SELECT [columns] FROM [table] WHERE [condition] ORDER BY [order];
CREATE TABLE parents AS
SELECT "abraham" AS parent, "barack" AS child UNION
SELECT "abraham" , "clinton" UNION
SELECT "delano" , "herbert" UNION
SELECT "fillmore" , "abraham" UNION
SELECT "fillmore" , "delano" UNION
SELECT "fillmore" , "grover" UNION
SELECT "eisenhower" , "fillmore";
CREATE TABLE dogs AS
SELECT "abraham" AS name, "long" AS fur UNION
SELECT "barack" , "short" UNION
SELECT "clinton" , "long" UNION
SELECT "delano" , "long" UNION
SELECT "eisenhower" , "short" UNION
SELECT "fillmore" , "curly" UNION
SELECT "grover" , "short" UNION
SELECT "herbert" , "curly";
```



First	Second
barack	clinton
abraham	delano
abraham	grover
delano	grover

```
SELECT a.child AS first, b.child AS second
FROM parents AS a, parents AS b
WHERE a.parent = b.parent AND a.child < b.child;
```

The number of groups is the number of unique values of an expression. A having clause filters the set of groups that are aggregated

```
select weight/legs, count(*) from animals
group by weight/legs
having count(*)>1;
```

weight/legs	count(*)
5	2
2	2

kind	legs	weight
dog	4	20
cat	4	10
ferret	4	10
parrot	2	6
penguin	2	10
t-rex	2	12000

```
CREATE TABLE ints(n UNIQUE, prime DEFAULT 1);
INSERT INTO ints VALUES (2, 1), (3, 1);
INSERT INTO ints(n) VALUES (4), (5), (6), (7), (8);
UPDATE ints SET prime=0 WHERE n > 2 AND n % 2 = 0;
DELETE FROM ints WHERE prime=0;
```

n	prime
2	1
3	1
5	1
7	1

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope).
 Lexical scope: The parent of a frame is the environment in which a procedure was defined. (lambda ...)
 Dynamic scope: The parent of a frame is the environment in which a procedure was called. (mu ...)

```
> (define f (mu (x) (+ x y)))
> (define g (lambda (x y) (f (+ x x))))
> (g 3 7)
13
```

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2, 3.3, true, +, quotient, ...
- Combinations: (quotient 10 2), (not true), ...

Numbers are self-evaluating; symbols are bound to values. Call expressions have an operator and 0 or more operands.

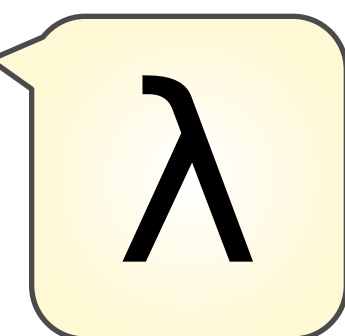
A combination that is not a call expression is a *special form*:

- If expression: (if <predicate> <consequent> <alternative>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <formal parameters>) <body>)

```
> (define pi 3.14)      > (define (abs x)
> (* pi 2)              (if (< x 0)
6.28                    (- x)
                        x))
> (abs -3)
3
```

Lambda expressions evaluate to anonymous procedures.

```
(lambda (<formal-parameters>) <body>)
```



Two equivalent expressions:

```
(define (plus4 x) (+ x 4))
(define plus4 (lambda (x) (+ x 4)))
```

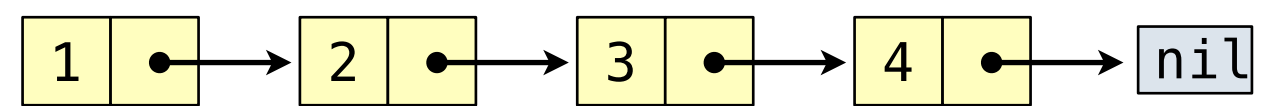
An operator can be a combination too:

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

In the late 1950s, computer scientists used confusing names.

- **cons**: Two-argument procedure that **creates a pair**
 - **car**: Procedure that returns the **first element** of a pair
 - **cdr**: Procedure that returns the **second element** of a pair
 - **nil**: The empty list
- They also used a non-obvious notation for linked lists.
- A (linked) Scheme list is a pair in which the second element is nil or a Scheme list.
 - Scheme lists are written as space-separated combinations.

```
> (define x (cons 1 (cons 2 nil)))
> x
(1 2)
> (car x)
1
> (cdr x)
(2)
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```



Symbols normally refer to values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of "a" and "b" in the resulting value

Quotation is used to refer to symbols directly in Lisp.

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Symbols are now values

Quotation can also be applied to combinations to form lists.

```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

```
class Pair:
    """A Pair has first and second attributes.

    For a Pair to be a well-formed list,
    second is either a well-formed list or nil.
    """
```

```
def __init__(self, first, second):
    self.first = first
    self.second = second
```

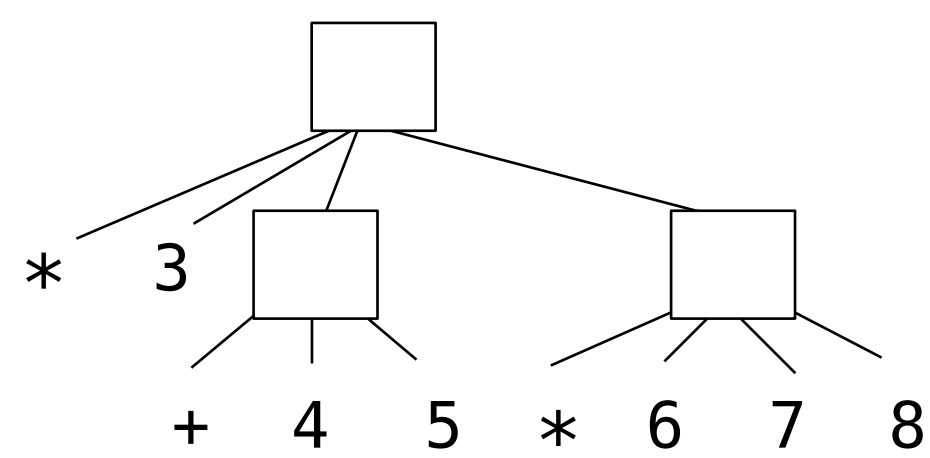
```
>>> s = Pair(1, Pair(2, Pair(3, nil)))
>>> print(s)
(1 2 3)
>>> print(Pair(1, 2))
(1 . 2)
>>> print(Pair(1, Pair(2, 3)))
(1 2 . 3)
>>> print(Pair(1, Pair(2, 3)).second)
(2 . 3)
```

The Calculator language has primitive expressions and call expressions

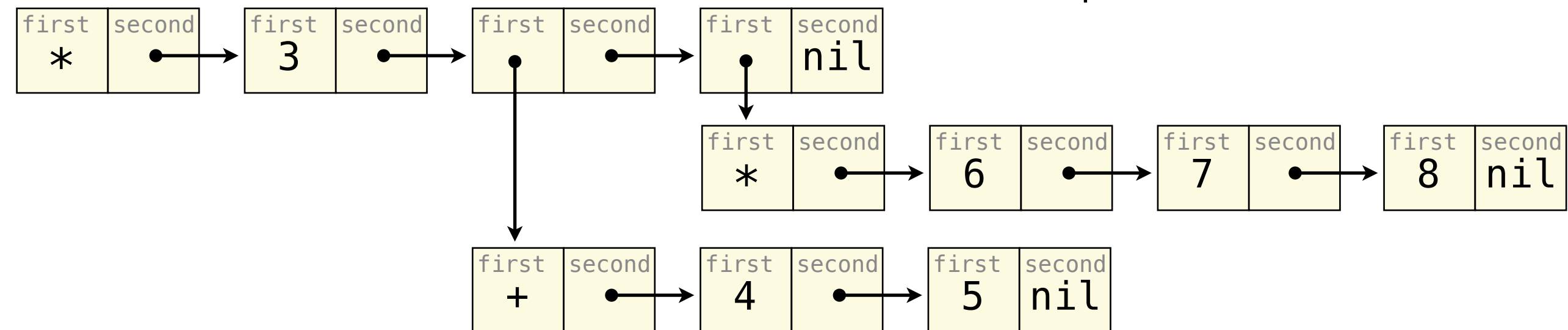
Calculator Expression

```
(* 3
 (+ 4 5)
 (* 6 7 8))
```

Expression Tree

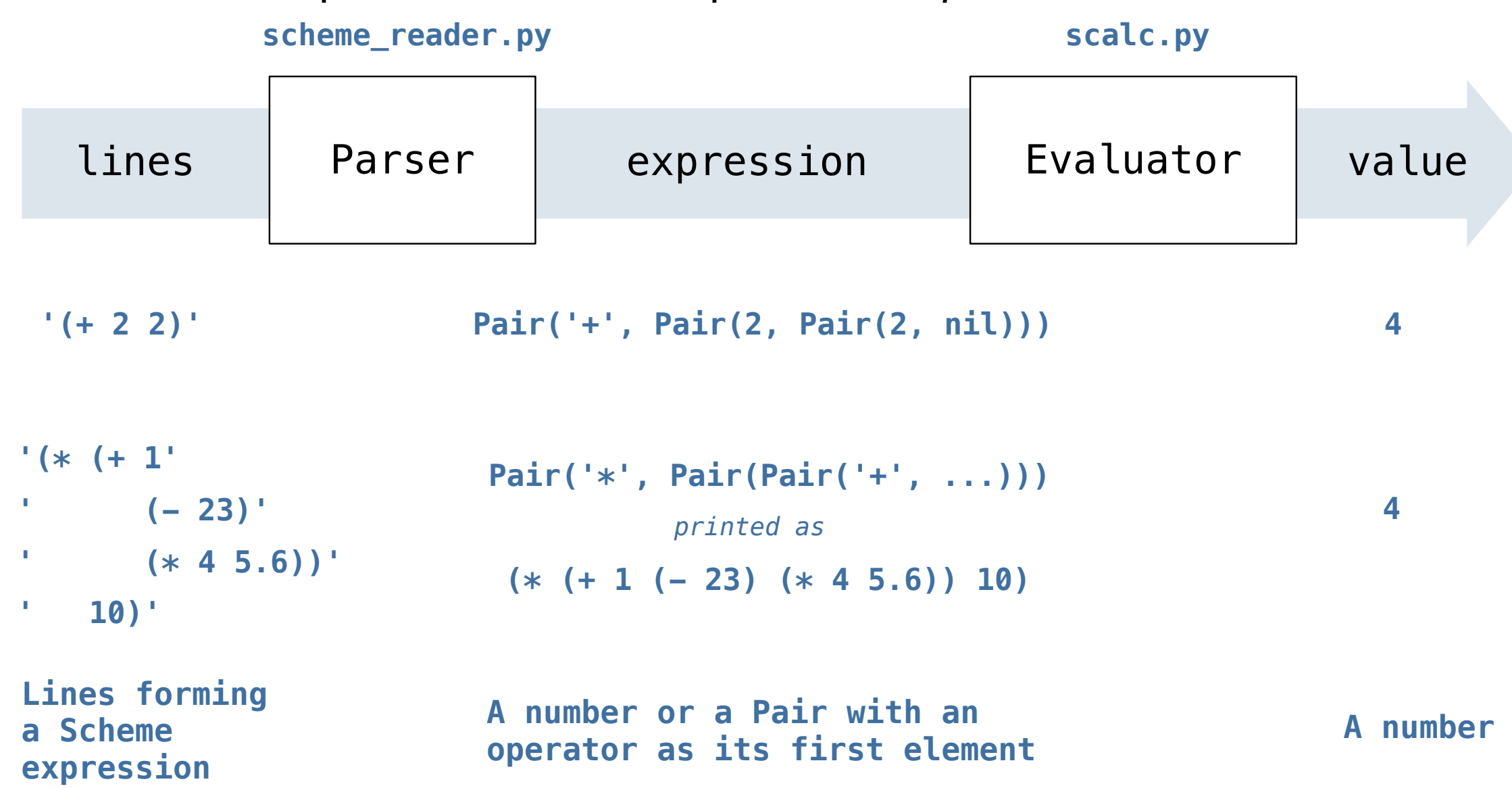


Representation as Pairs

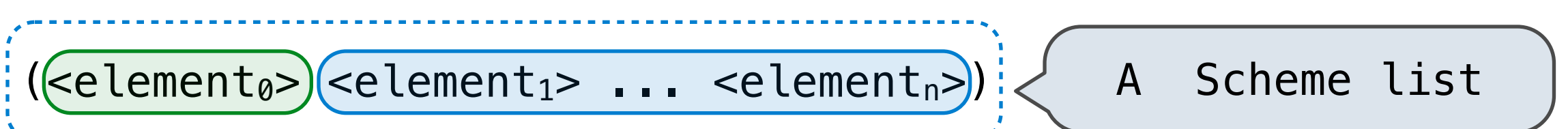


- Recursive decomposition: `def count_partitions(n, m):` finding simpler instances of a problem.
- E.g., `count_partitions(6, 4)`
- Explore two possibilities:
 - Use at least one 4
 - Don't use any 4
- Solve two simpler problems:
 - `count_partitions(2, 4)` → with_m = count_partitions(n-m, m)
 - `count_partitions(6, 3)` → without_m = count_partitions(n, m-1)
- Tree recursion often involves exploring different choices.

A basic interpreter has two parts: a *parser* and an *evaluator*.



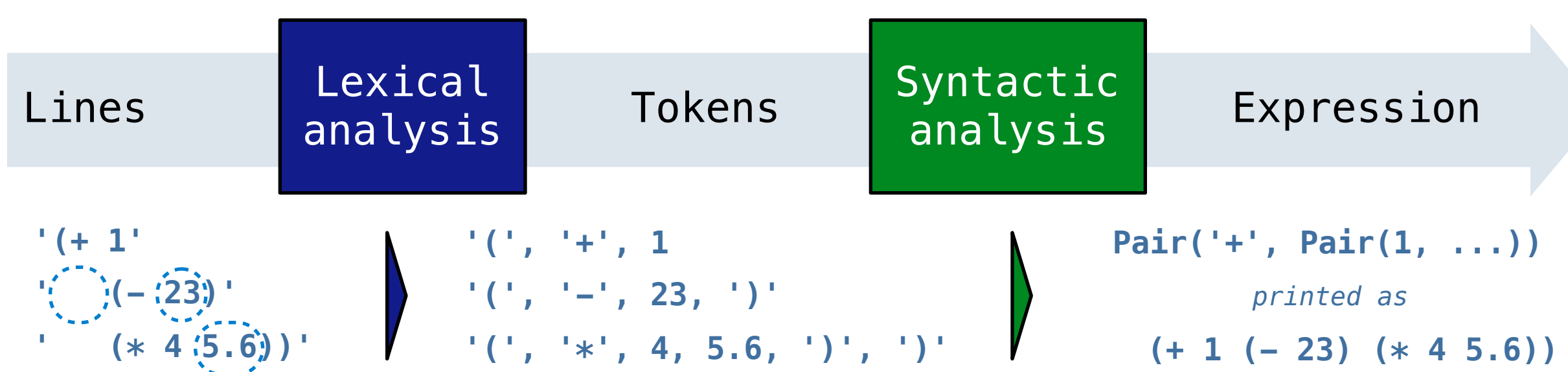
A Scheme list is written as elements in parentheses:



Each <element> can be a combination or atom (primitive).
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))

The task of *parsing* a language involves coercing a string representation of an expression to the expression itself. Parsers must validate that expressions are well-formed.

A Parser takes a sequence of lines and returns an expression.



- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

- Tree-recursive process
- Balances parentheses
- Returns tree structure
- Processes multiple lines

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to `scheme_read` consumes the input tokens for exactly one expression.

Base case: symbols and numbers

Recursive call: `scheme_read` sub-expressions and combine them

- Base cases:**
- Primitive values (numbers)
 - Look up values bound to symbols
- Recursive calls:**
- Eval(operator, operands) of call expressions
 - Apply(procedure, arguments)
 - Eval(sub-expressions) of special forms

Eval

The structure of the Scheme interpreter

Creates a new environment each time a user-defined procedure is applied

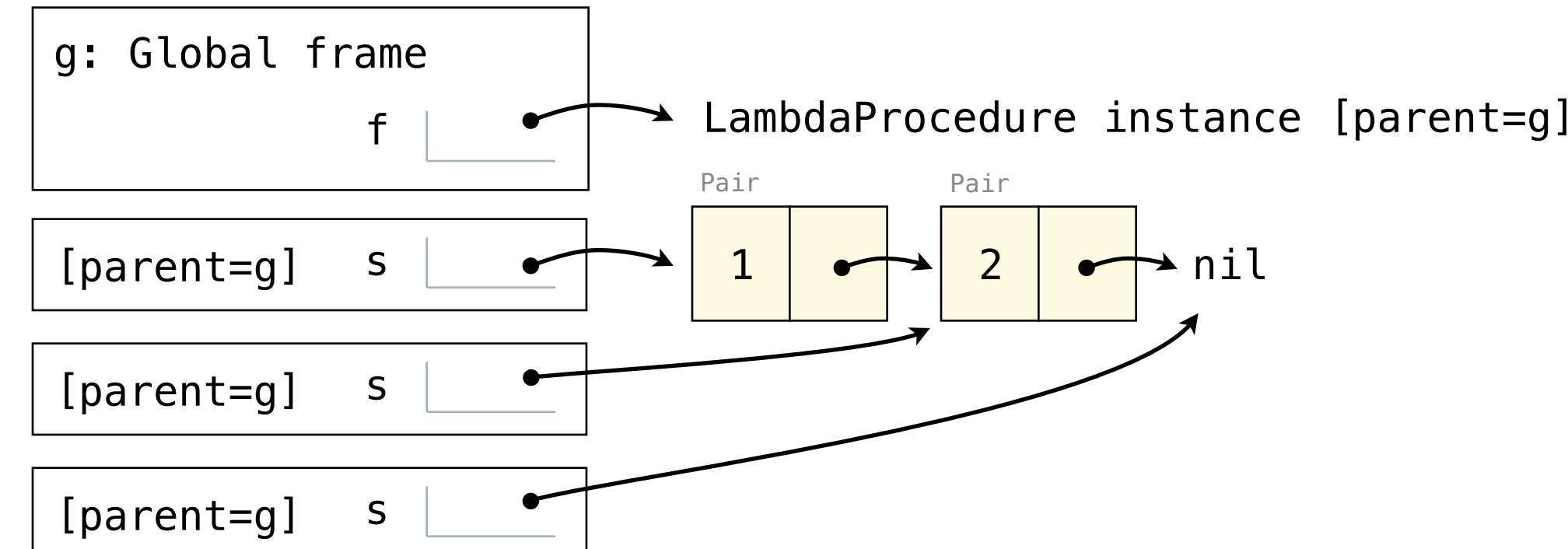
Requires an environment for name lookup

- Base cases:**
- Built-in primitive procedures
- Recursive calls:**
- Eval(body) of user-defined procedures

Apply

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure, then evaluate the body of the procedure in the environment that starts with this new frame.

```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))
(f (list 1 2))
```



How to Design Functions:

- 1) Identify the information that must be represented and how it is represented. Illustrate with examples.
- 2) State what kind of data the desired function consumes and produces. Formulate a concise answer to the question *what* the function computes.
- 3) Work through examples that illustrate the function's purpose.
- 4) Outline the function as a template.
- 5) Fill in the gaps in the function template. Exploit the purpose statement and the examples.
- 6) Convert examples into tests and ensure that the function passes them.