CS 61A Final Exam Study Guide – Page 1

Exceptions are raised with a raise statement.

```
raise <expression>
```

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

Exceptions are constructed like any other object. E.g.,
TypeError('Bad argument!')

```
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

If the class of the exception inherits from <exception class>, then

The <except suite> is executed, with <name> bound to the exception.

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

handling a <class 'ZeroDivisionError'>
>>> x
0
```

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header <expression>, which yields an iterable object.
2. For each element in that sequence, in order:
   A. Bind <name> to that element in the first frame of the current environment.
   B. Execute the <suite>.

An iterable object has a method __iter__ that returns an iterator.

```
>>> counts = [1, 2, 3]
>>> for item in counts:
        print(item)
1
2
3
```

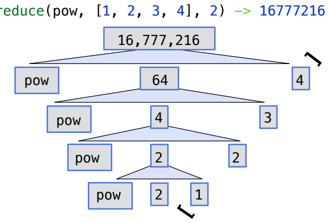
```
>>> items = counts.__iter__()
>>> try:
        while True:
            item = items.__next__()
            print(item)
    except StopIteration:
        pass
```

```
map(func, iterable):          Iterate over func(x) for x in iterable
filter(func, iterable):       Iterate over x in iterable if func(x)
zip(first_iter, second_iter): Iterate over co-indexed (x, y) pairs
```

```
def reduce(f, s, initial):
    """Combine elements of s pairwise
    using f, starting with initial.

    >>> reduce(mul, [2, 4, 8], 1)
    64
    """
    for x in s:
        initial = f(initial, x)
    return initial
```

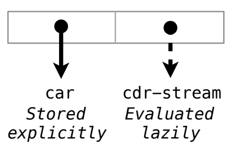
reduce(pow, [1, 2, 3, 4], 2) -> 16777216




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

(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
```

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




```
(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))              1   1   1 ...
(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)))   1   2   3 ...
```

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

```
(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 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
```

```
iter(iterable):
  Return an iterator
  over the elements of
  an iterable value

next(iterator):
  Return the next element
  in an iterator
```

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> next(iter(s))
3
```

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> k = iter(d)          >>> v = iter(d.values())
>>> next(k)              >>> next(v)
'one'                    1
>>> next(k)              >>> next(v)
'three'                  3
>>> next(k)              >>> next(v)
'two'                    2
```

A **StopIteration** exception is raised whenever **next** is called on an empty iterator

```
>>> contains('strength', 'stent')
True
>>> contains('strength', 'rest')
False
def contains(a, b):
    ai = iter(a)
    for x in b:
        try:
            while next(ai) != x:
                pass # do nothing
        except StopIteration:
            return False
    return True
```

A *generator function* is a function that **yield**s values instead of **return**ing them
A normal function **returns** once; a *generator function* can **yield** multiple times
A *generator* is an iterator created automatically by calling a *generator function*
When a *generator function* is called, it returns a *generator*

```
>>> def plus_minus(x):
...     yield x
...     yield -x
>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
```

```
class Countdown:
    def __init__(self, start):
        self.start = start
    def __iter__(self):
        v = self.start
        while v > 0:
            yield v
            v -= 1
```

```
>>> list(Countdown(5))
[5, 4, 3, 2, 1]
>>> for x in Countdown(3):
...     print(x)
3
2
1
```

A **yield from** statement yields all values from an iterator or iterable

```
>>> list(a_then_b([3, 4], [5, 6]))
[3, 4, 5, 6]
```

```
def a_then_b(a, b):           def a_then_b(a, b):
    for x in a:          OR       yield from a
        yield x                   yield from b
    for x in b:
        yield x
```

A **table** has columns and rows

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

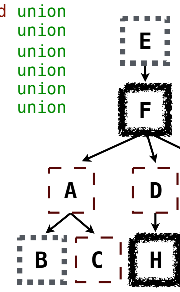
A **column** has a name and a type

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";
```




```
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;
```

First	Second
barack	clinton
abraham	delano
abraham	grover
delano	grover

```
with
  ancestors(ancestor, descendent) as (
    select parent, child from parents union
    select ancestor, child
           from ancestors, parents
           where parent = descendent
  )
select ancestor from ancestors where descendent="herbert";
```

ancestor
delano
fillmore
eisenhower

```
create table pythagorean_triples as
  with
    i(n) as (
      select 1 union select n+1 from i where n < 20
    )
  select a.n as a, b.n as b, c.n as c
         from i as a, i as b, i as c
         where a.n < b.n and a.n*a.n + b.n*b.n = c.n*c.n;
```

a	b	c
3	4	5
5	12	13
6	8	10
8	15	17
9	12	15
12	16	20

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

weight/legs=5
weight/legs=2
weight/legs=2
weight/legs=3
weight/legs=5
weight/legs=6000

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

CS 61A Final Exam Study Guide – Page 2

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)
> (* pi 2)
6.28
```

```
> (define (abs x)
    (if (< x 0)
        (- x)
        x))
> (abs -3)
3
```

Lambda expressions evaluate to anonymous procedures.

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

λ

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.
- A dotted list has an arbitrary value for the second element of the last pair. Dotted lists may not be well-formed lists.

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

Not a well-formed list!

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.

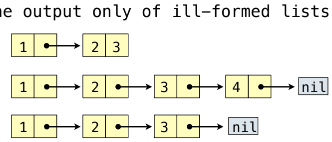
```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

Dots can be used in a quoted list to specify the second element of the final pair.

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists.

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)
> (cdr '((1 2) . (3 4 . (5))))
(3 4 5)
```




```python
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
```

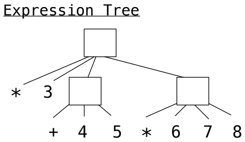
```
>>> s = Pair(1, Pair(2, Pair(3, nil)))
>>> print(s)
(1 2 3)
>>> len(s)
3
>>> print(Pair(1, 2))
(1 . 2)
>>> print(Pair(1, Pair(2, 3)))
(1 2 . 3)
```

The Calculator language has primitive expressions and call expressions

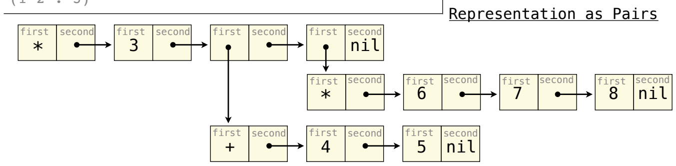
Calculator Expression

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

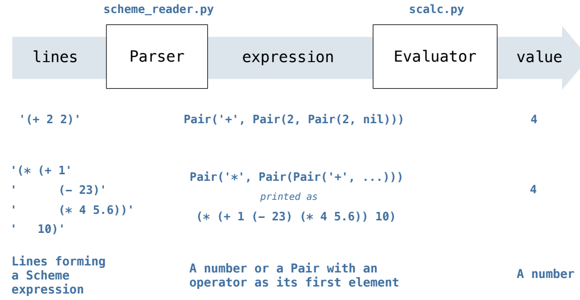
Expression Tree



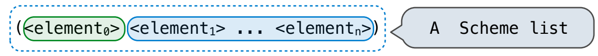
Representation as Pairs



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



A Scheme list is written as elements in parentheses:

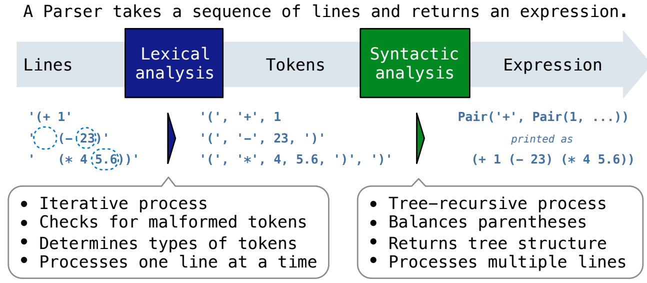


Each <element> can be a combination or atom (primitive).
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))

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.

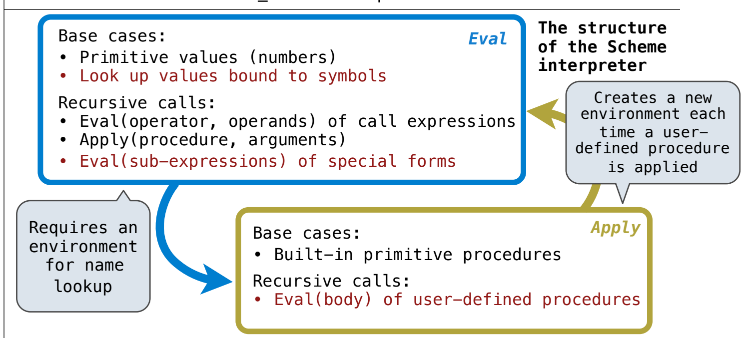


Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

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

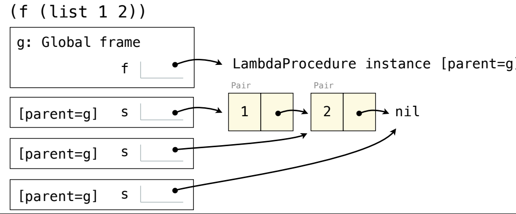
Base case: symbols and numbers
Recursive call: scheme_read sub-expressions and combine them

The structure of the Scheme interpreter



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))
```



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** expression

```
(define (factorial n k)
  (if (= n 0) k
    (factorial (- n 1)
               (* k n))))
```

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)))))
```

Not a tail call

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

Recursive call is a tail call