**CS 61A Final Exam Fall 2020 Study Guide – Page 1**

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

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
```

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
```

There are two ways to quote an expression

```
Quote:      '(a b)   =>   (a b)
Quasiquote: `(a b)   =>   (a b)
```

They are different because parts of a quasiquoted expression can be unquoted with ,

```
            (define b 4)
Quote:      '(a ,(+ b 1))  =>   (a (unquote (+ b 1))
Quasiquote: `(a ,(+ b 1))  =>   (a 5)
```

Quasiquotation is particularly convenient for generating Scheme expressions:

```
(define (make-add-procedure n) `(lambda (d) (+ d ,n)))
(make-add-procedure 2)  => (lambda (d) (+ d 2))
```

```
; Sum the squares of even numbers less than 10, starting with 2
; x = 2
; total = 0
; while x < 10:
;     total = total + x * x
;     x = x + 2
; RESULT: 2 * 2 + 4 * 4 + 6 * 6 + 8 * 8 = 120

(begin
  (define (f x total)
    (if (< x 10)
      (f (+ x 2) (+ total (* x x)))
      total))
  (f 2 0))

; Sum the numbers whose squares are less than 50, starting with 1
; x = 1
; total = 0
; while x * x < 50:
;   total = total + x
;     x = x + 1
; RESULT: 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28

(begin
  (define (f x total)
    (if (< (* x x) 50)
      (f (+ x 1) (+ total x))
      total))
  (f 1 0))

(define   (sum-while starting-x while-condition add-to-total update-x)
  ; (eval (sum-while 2          '(< x 10)       '(* x x)     '(+ x 2)))  =>  120
  ; (eval (sum-while 1          '(< (* x x) 50) 'x           '(+ x 1)))  =>  28
  `(begin
     (define (f x total)
       (if ,while-condition
         (f ,update-x (+ total ,add-to-total))
         total))
     (f ,starting-x 0)))
```

```
(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 nil) size 'size) ; => ((1) 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)
```

```
;; 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 **row** has a value for each column

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

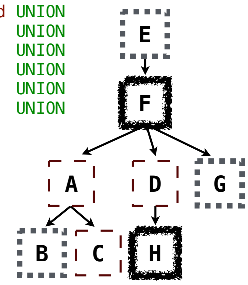
```
SELECT [expression] AS [name], [expression] AS [name], ... ;

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 |

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 |



Optional content removed

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 nil))
> x
(1)
> (car x)
1
> (cdr x)
()
> (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.

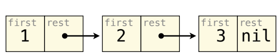
```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

```
(car (cons 1 nil)) -> 1
(cdr (cons 1 nil)) -> ()
(cdr (cons 1 (cons 2 nil))) -> (2)
```

```
class Pair:
    """A pair has two instance attributes:
      first and rest.

      rest must be a Pair or nil.
    """
    def __init__(self, first, rest):
        self.first = first
        self.rest = rest

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

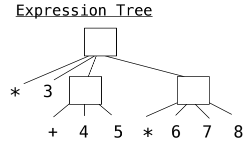



The Calculator language has primitive expressions and call expressions

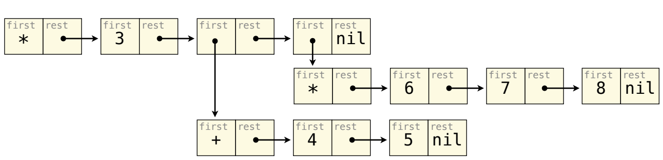
Calculator Expression

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

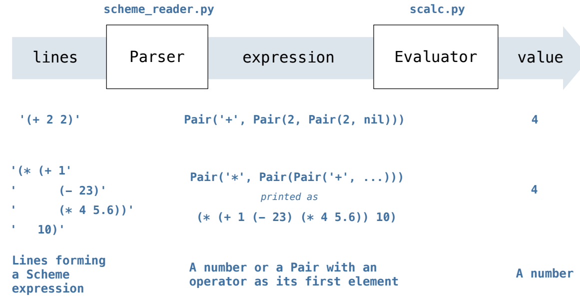
Expression Tree



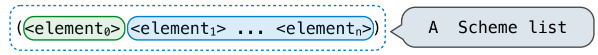

Representation as Pairs




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




A Scheme list is written as elements in parentheses:



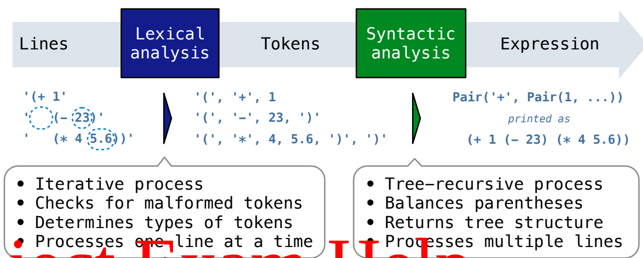

Each <element> can be a combination or atom (primitive).

(+ (* 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.

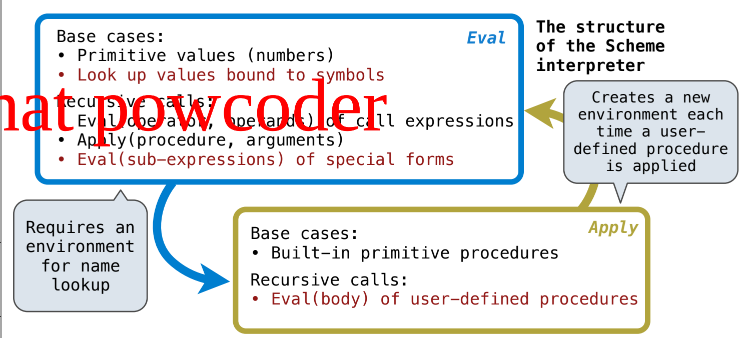



Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

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

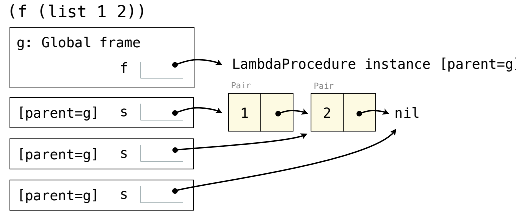
**Base case:** symbols and numbers
**Recursive call:** scheme_read sub-expressions and combine them

**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))
```




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.